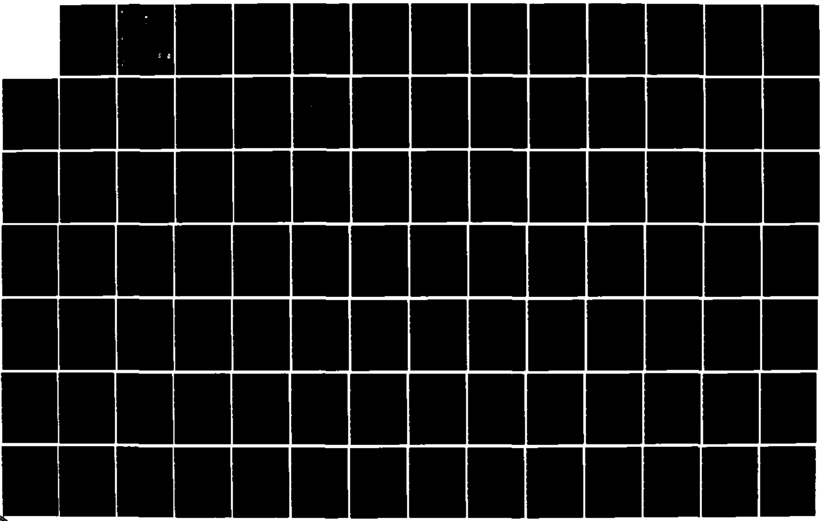
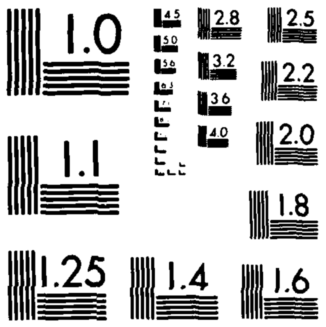
AD-A151 770 IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G KEARNS 1/2

UNCLASSIFIED DEC 84 AFIT/GCS/ENG/84D-12 F/G 9/2 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

1

IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM

THESIS

Timothy G. Kearns
Captain, USAF

AFIT/GCS/ENG/84D-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

⓪

AFIT/GCS/ENG/84

IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM

THESIS

Timothy G. Kearns
Captain, USAF

AFIT/GCS/ENG/84D-12

AFIT/GCS/ENG/84D-12

IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Timothy G. Kearns, B.S.

Captain, USAF

December 1984

Preface

The desire for an in house relational database for teaching and research purposes was expressed by Dr. Thomas Hartrum, on the faculty of the AFIT/EN Electrical Engineering Department. Thus, in 1979, Mark A. Roth started the initial development of such a database. He left a partially developed system and recommendations of how to complete the system. In 1982, Linda M. Rodgers continued the development of the database system. She extended the implementation of the system but left the system still not completely implemented. I undertook this project with the intention of completing the implementation of the system and analyzing the performance of the system.

The initial goal had to be revised to just providing an operational system when it was determined that even using numerous memory overlays the system was just too large to efficiently operate on the computer being used. Also, the memory overlays caused numerous unexpected problems that greatly hampered the development. Hopefully, the recommendations, design and implementation provided will allow the system to be easily expanded to its full capability.

I owe many thanks to Dr. Hartrum for his comments, advice, encouragement, and understanding during this thesis effort. Thanks are also due to Dr. Potoczny for his advice and comments and Dan Zambon and Charlie Powers of the AFIT/ENE staff for their assistance.

Table of Contents

List of Figures

Abstract

The single processor optimized relational database system is a database system designed and implemented for teaching and research purposes at the Air Force Institute of Technology. The system was originally designed and partially implemented by Mark A. Roth in 1979. The design and implementation was continued by James Mau in 1981 and Linda M. Rodgers in 1982. To complete the implementation of the relational database system, an investigation of the design and implementation of the previous research efforts was done. Additional research was done to explore possible designs and implementations of access structures and possible methods to implement the relational operators.

With this background, a structured design was completed for the access structure and the relational operators. Once this was accomplished, the low level access structure was implemented and tested, providing the capability to insert, delete, and modify data in the relational database system. Finally, some of the relational operators were implemented and tested providing an operational relational database system.

IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM

I. Introduction

There is an ever increasing demand for a better way to handle the vast amounts of data necessary in today's computer world. Research has provided database management systems to aid in the management of this information. Although database management systems have helped to alleviate the problem of managing information, most database management systems are complex and require a computer professional to understand and use them.

The introduction of the relational database model provided the basis for a database system that the users could easily understand and use. The fact that the user can look at the information in the logical manner in which he wants to see it, not the way the information is stored in the computer, causes the "user-friendly" environment. But, this "user-friendly" environment causes some significant problems for the database management system in storing and handling the information. The most significant problem is the complexity of handling the data logically. This causes the relational database system to be slow to respond to the user.

Very few relational database systems are commercially available because of the complexity of handling the data.

Thus, when the AFIT community expressed a desire to install a relational database system for teaching and research purposes, the lack of available relational database systems caused a concern of how to handle the problem. This concern started research to develop and implement a pedagogical relational database system.

The design of the AFIT database has taken into consideration the time problem by using query optimization techniques (6) and the data retrieval and accessing problem by using a B-tree concept in the file handling (5). The research efforts have been fruitful in these areas but they have never been fully realized because the AFIT relational database still lacks the low-level access structure necessary to make the system operational. Until the total system can be fully implemented, an accurate measure of how successful the AFIT relational database system is at alleviating the inherent problems of a relational database can not be completed.

Statement of Problem

The purpose of this thesis was to complete two tasks. First, the remaining access structure of the AFIT relational database system was completed, allowing data to inserted into the data base. Second, the main relational operators were implemented to provide the ability to retrieve data from the database. This also will provide the ability to evaluate the performance of the query optimization concept.

Scope and Assumptions

The scope of this thesis was to complete the implementation of the AFIT relational database system. To complete the implementation, it was assumed that the previous work by Roth and Rodgers was correct with only minor modifications. The modifications included alleviating the memory space problem that had arisen in the development effort by converting to a PASCAL that provided more management of the computer's memory instead of the form of PASCAL now used. Next, the design and implementation of the remaining access structure modules was completed. Then the relational operators were coded and tested to make the system operational.

General Approach

The first step in solving this problem was a review of the literature from which the AFIT relational database originates. It included reviewing literature on the query optimization techniques and the file structure implementation. Next, the actual code was examined. This was done for a twofold purpose, first to understand how the database is currently implemented and secondly for the purposes of converting the code from the current form of UCSD Pascal to Pascal/MT+.

After becoming familiar with what existed, the next step was converting the existing software to Pascal/MT+ to provide better utilization of the computer's memory space. Pascal/MT+ was selected as the Pascal compiler because of preserving transformations. This essentially means to push the selects and projects as low as possible in the query tree thus requiring less data to be stored and manipulated in temporary relations. Also, by pushing the selects and projects down to the lowest level, the capability to utilize the B-tree access structure for random processing greatly increases. The utilization of the directory of the access structure should provide faster processing time because the amount of data to be manipulated and examined should be greatly reduced.

The tree transformer also optimizes the query tree by combining subtrees that contain set operators on a common relation into a single compound operation on the relation (7: 572). This causes the relation to only be read once instead of having multiple reads of the relation. Also, if directories exist for all the domains involved, it might only be necessary to examine the directories to provide pointers to the desired tuples, thus eliminating even more accesses.

The tree transformer reduces the amount of memory space necessary for storing temporary relations by using the correctness preserving transformations. It then tries to reduce the amount of processing time by reducing the processing of redundant data by combining operations on a common relation. The output from the tree transfomer is a optimized query tree. This optimization alone should be an improvement over the unoptimized query, but the coordinating

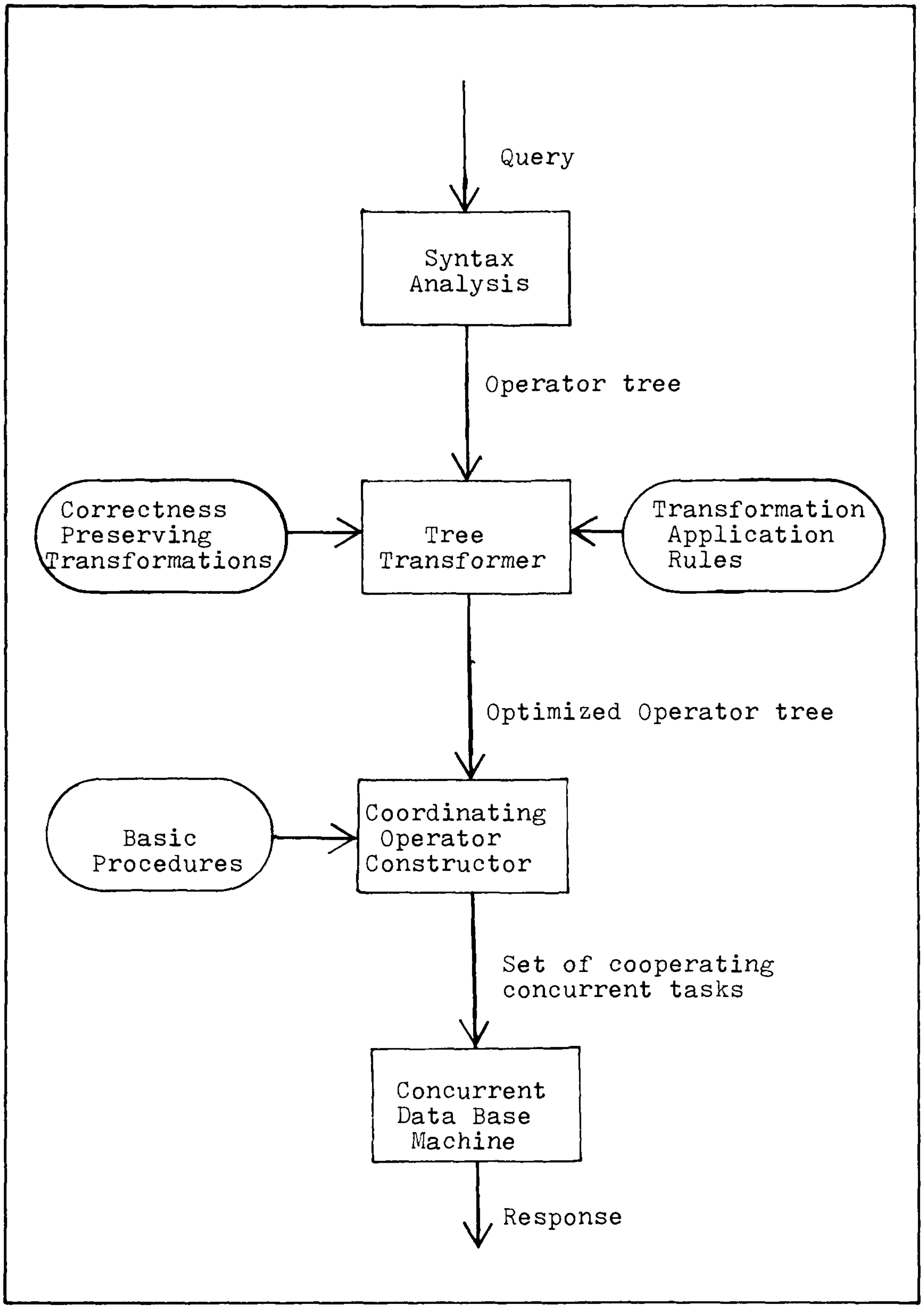

Query
Syntax Analysis
Operator tree
Correctness Preserving Transformations
Tree Transformer
Transformation Application Rules
Optimized Operator tree
Basic Procedures
Coordinating Operator Constructor
Set of cooperating concurrent tasks
Concurrent Data Base Machine
Response

Figure 6. The basic organization of the query optimizer

change, or delete tuples from a relation, and the run modules which actually provide the ability to query the information contained in the database.

The run modules allow for queries to be constructed and edited before they are executed. The queries are constructed using a form of relational algebra. An option exists to allow the user to save the command file that contains the queries for later use. Once the command file has been created and execution started, the queries are checked for the correct syntax and then optimized utilizing the principles formulated by Miles Smith and Philip Chang (7).

Query Optimization. There are two types of query optimization: an optimization of the low level efficiency of access paths and the efficiency of disk accessing; or a high level optimization by transforming the query into a more efficient structure so that the work expended by the low level access structures is kept to a minimum. The primary concern of the work done before this thesis was concerned with the high level transformations of queries. The following sections briefly describe the steps taken to optimize queries and how the steps were implemented at the start of this thesis effort. Figure 6 portrays the basic organization of the query optimizer.

Tree Transformer. The tree transformer receives the syntactically correct queries in the form of a query tree. This query tree is then transformed using correctness functions to shrink the tree as necessary to keep it in the balanced form of a B-tree.

The B-tree insertion functions allow the adding of new values to the B-tree. If this causes overflow, then a new B-tree node is created and added to the B-tree and the height of B-tree is adjusted. The insertion of a new domain value can cause several modules to be called that add and split nodes. The new value added to the B-tree points to a leaf where the tuple identifiers are stored. So, the purpose of the B-tree nodes is to provide an index to the proper leaf. The storing of the tuple identifiers in the leaves and providing links between the leaves provides for the tuples to be accessed sequentially using only the leaf nodes.

The modules that delete tuple identifiers from the existing B-tree provide for keeping the tree balanced by combining the index nodes as necessary and even creating a new root for the tree if necessary. The deletion of a tuple identifier only affects the B-tree if the value deleted was the largest value in the leaf. It then causes the value to be deleted from the B-tree. Rodgers developed and tested the B-tree modules described above and tested them as stand-alone modules but never could incorporate them as fully integrated parts of the system due to a lack of memory space (5).

The data manipulation facilities include the edit module, which uses the access structure described above to add, B-tree Record

Fields in B-tree Record

A	B	C	D	E		
				E1	E2	E3
1	1	3	Variable	1	15	3

Length in Characters

Field Definitions:

A - Deletion Field: A field that indicates whether or not the record has been deleted.

B - Record Type: A 'B' is this field indicates that this is a B-tree record.

C - Number of Keys: The number of keys currently in this B-tree node.

D - Domain Value: The domain value that is the largest value in the leaf or child node that this record points to.

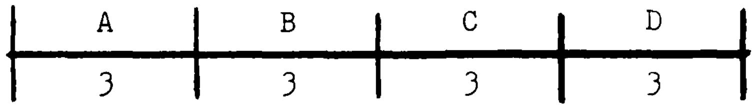
E - Pointer Field: A pointer to either a child B-tree node or a leaf node.

- E1 - The pointer type field. 'B' in this field indicates the pointer points to a child node in the B-tree and 'L' indicates it points to a leaf node.
- E2 - Filename of the leaf or node pointed to.
- E3 - Block number of node or leaf pointed to.

Figure 5. B-tree Record Format

B-tree Header Record

Fields in B-tree Header Record



Length in Characters

Field Definitions:

A - B-tree root: The block number in the B-tree that contains the B-tree root.

B - Maximum number of keys in leaf. The maximum number of keys in a leaf node.

C - Maximum number of nodes in a leaf.

D - Fanout ratio: The fanout ratio or maximum number of records in a B-tree node. Computed by dividing the size of one block by the size of one B-tree record. It can range between 4 and 10.

Figure 4. B-tree Header Record Format.

the access structure. There are many different access paths and associated structures but the one used in this system is based on a B-tree (5). The original structure recommended by Roth (6) was based on Theo Haerder's generalized access path structure (1). Rodgers then partially implemented the concept of the generalized access path using the B-tree concept (5).

The B-tree concept utilizes a B+-tree to provide an index to the leaf structure. The leaf structure contains the tuple identifiers (TIDs). The TIDs can then be used to directly access the tuples in the relations. To facilitate operations on the attributes, a B-tree and leaf structure are maintained for each non-text domain. The values of all attributes defined on the same domain are stored in the access structure for that domain.

The B-tree structure developed has two record formats. The first is the B-tree header record. The B-tree header record provides the number of B-tree records that can fit in a node of the B-tree and what block of the B-tree file contains the root of the B-tree. Figure 4 shows the format of B-tree header record and Figure 5 shows the B-tree record.

The B-tree modules that were developed provided the ability to determine if a B-tree currently exists for the domain and the capability of inserting and deleting individual values from the B-tree. The insertion of a value into the B-tree requires several functions to expand and split the tree as necessary. The deletion modules provide the background for the continued development of the system.

Data Input, Modification, and Deletion.

Although the edit facility , which contains the input, modify, and delete modules, was not completely implemented the access structure modules are in place. The biggest factor in the data storage and manipulation facility is the access structure. This was important since it is assumed that the amount of information to be stored in the database is too great to be stored in primary memory. Also, any information stored in primary memory would be lost as soon as the computer was shut off.

The Edit module controls the input, modification, and deletion of data in the database. These modules either use the access structure to determine where the data is stored or update the access structure with the location of new data being added in the system. The following sections describe the access structure and the modules that existed to manipulate the structure.

Access Structure. The access structure is the map of where the data is located in secondary memory. This mapping is used whenever data is needed. It allows the database management system to find the information in secondary memory and then load the data into main memory for processing. Since the operation of the database management system depends on the access structure, the performance of the system depends a great deal on the efficiency and effectiveness of

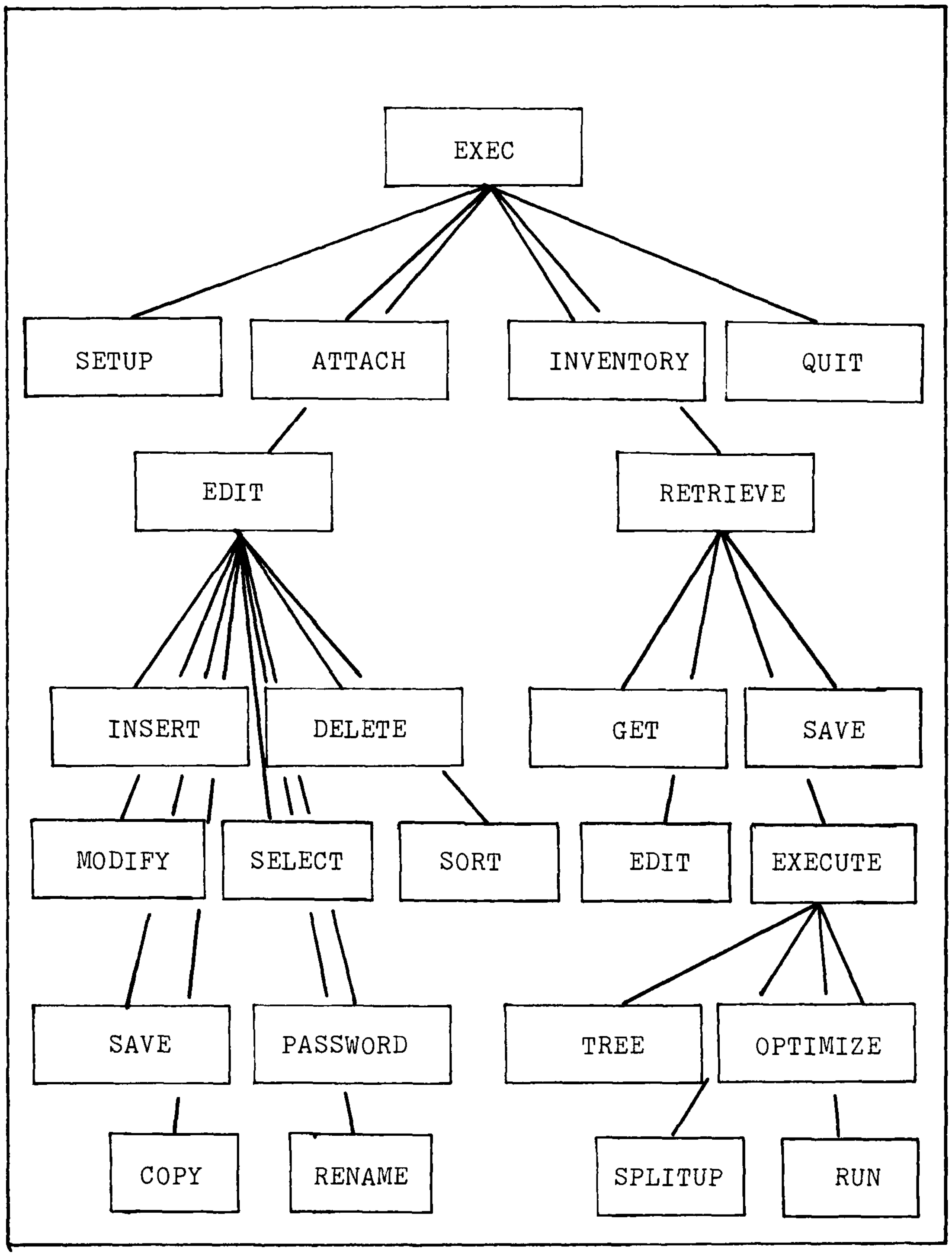


Figure 3. The DML Processor

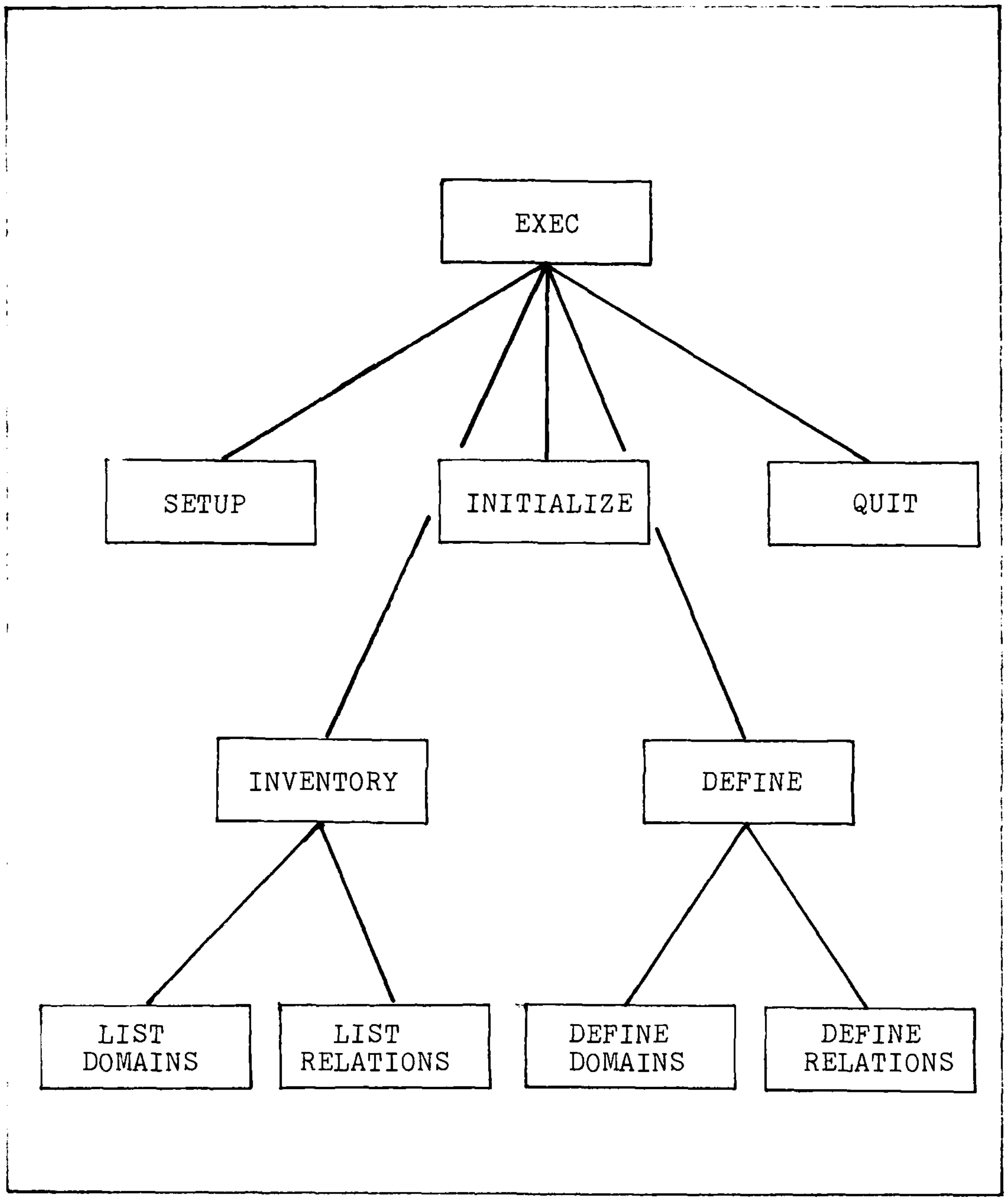


Figure 2. The DDL Processor
Modified from Figure 2 of Reference 5

resources, divided the system into two major areas: data definition and data manipulation. Figure 2 reflects the data definition facility and Figure 3 shows the data manipulation facility. The following sections discuss some of the major portions of the database system.

Data Definition

The data definition module provides the capability to define domains and relations. This facility was originally implemented as part of the complete system (6: 43-45). During later development of the system, it was split off and made a standalone module of the system (5: 40-42). The data definition module was made a standalone part of the system to allow only the database administrator the ability to define new domains and relations and the capability to destroy current definitions of domains and relations. This provides for better integrity for the system and provides centralized control of the definition process. Figure 2 shows the data definition facility.

Data Manipulation

The data manipulation actually can be divided into two areas: data input, modification, and deletion; and query optimization and data retrieval. This allows the user of the database to enter data into the database, manipulate it as necessary, and retrieve it in response to queries. Neither facility was completely implemented but the existing modules and considerations will be discussed to provide a

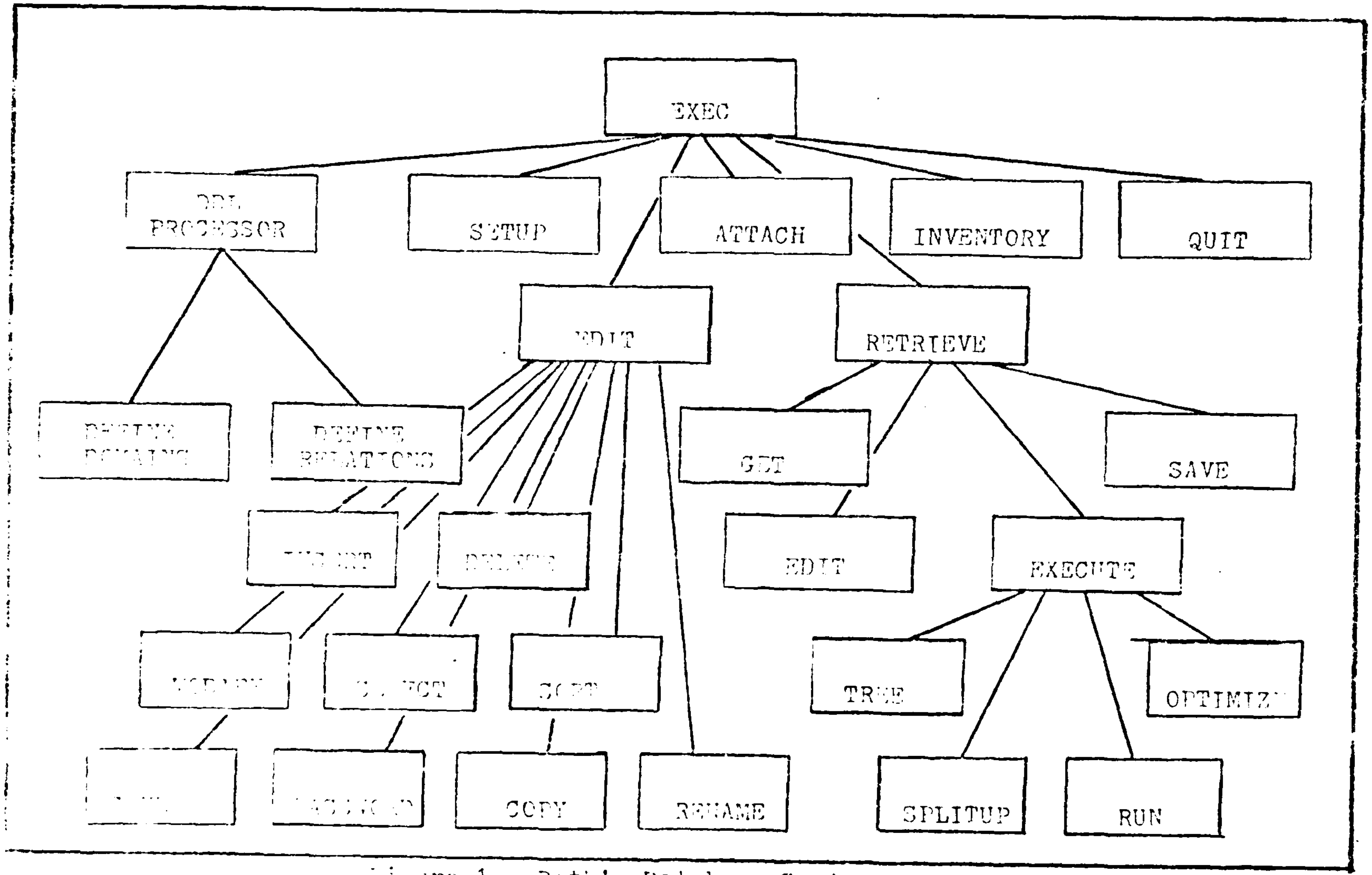


Figure 1. Roth's Database System

II. Background

E. F. Codd first introduced the idea of utilizing the relational concept for data in 1970. Since that time, much has been written about the theoretical concepts of the relational view. Not only have the theoretical aspects of the relational view been studied but also the practical aspects have been widely researched. Some of the current implementations of relational database are Ingres, Query By Example, and System R (3: 187-195). The advent of the microcomputer caused a demand for a relational database system that will run efficiently on this type machine. Some relational-like databases, such as dBase II and Condor, have become available for a microcomputer but these systems were not readily available in 1979. Therefore, in 1979, Mark A. Roth started the design and implementation of a relational database system for a microcomputer (6: 8-10).

Roth considered many key aspects of the relational database in his design and implementation. Some of these aspects were the means for data definition and data manipulation. The original system was designed and partially implemented on the Intel Series II (6). Figure 1 shows the state of the system at the conclusion of Roth's thesis effort. Implementation was continued by Mau on the LSI-11 using UCSD Pascal (2). Linda M. Rodgers continued the design and implementation of the relational database in 1982 (5). Rodgers, in an effort to better utilize limited its availability and its ability to provide program overlays. Also, Pascal/MT+ statements very closely resemble UCSD Pascal, thus making the conversion somewhat easier. The code was then converted to the new form of Pascal.

After the original software was converted, the design and implementation of the remaining modules began. A structured design approach was used to design the remaining modules which include the low-level access modules, portions of the edit modules, and the run modules. The modules were coded and tested independently before they were integrated into the complete system. Finally, a system test was executed to validate the integration and operation of the complete system.

Sequence of Presentation

The presentation of this thesis parallels the approach to solving this problem. Chapter II provides background information used in the previous and current development of the AFIT Relational Database. Chapter III presents the details of the design and implementation of the remaining modules and Chapter IV contains the description of the testing done to verify and validate the system. Finally, Chapter V presents conclusions and recommendations.

operator constructor provides even more optimization.

The Coordinating Operator Constructor. The coordinating operator constructor takes the transformed query tree provided by the tree transformer and implements each operator represented from a basic set of procedures in such a way that the sort orders of the intermediate relations are optimally cooordinated. By providing optimal sort orders for the intermediate relations, the search time is reduced if directories exist for the domains chosen as sort orders. Also, if the sort order chosen is a primary key, processing time is reduced because the operation can use the directory for the domain and not have to eliminate duplicate tuples.

The coordinating operator constructor makes two passes over the operator tree. On the first pass, an upward pass, each branch is labeled with a preferred sort order. The second pass is a downward pass that examines the set of perferred sort orders and makes a final decision on which preferred sort order will be implemented (7: 573-576).

Roth implemented the query optimization technique described above but also included some additional features. The additional features include not only optimizing a single expression as was considered by Smith and Chang, but also optimizing multiple queries. Multiples queries means queries where the user expects several relations returned as the result. The additional features of Roth's implementation are concentrated in two modules. The first module, the tree module, examines subtrees in different queries to determine if subtrees might be shared. The output of this module is a network of shared trees. The other module, the splitup module, produces an order-optimized forest of separate trees in which all the shared subtrees have been removed (6: 55). At the start of this thesis effort the coordinating operator constuctor was not fully implemented.

Summary.

The implementation of the AFIT relational database by Mark Roth and Linda Rodgers has addressed several key aspects associated with a relational database system. The data definition facility, included to allow the database administrator the ability to define and control all relations and domains, is one of the key concepts of a database system that has been implemented. The next concept of a relational database system that was implemented was the access path and directory structure developed by Rodgers to allow for more efficient access to the data stored on secondary memory. And the final concept that was implemented was the optimization techniques implemented by Roth to provide faster response time to queries of data in the database system. Although these structures and techniques are key concepts of a relational database system and have been implemented, the system still lacked the design and implementation ofthe procedures to input, modify, and delete data from the database and all of the relational operators to retrieve the data from the database.

III. Design and Implementation

The design of the remaining parts of the relational database system had four primary tasks: conversion of existing software from UCSD Pascal on an LSI-11 to Pascal/MT+ on a Z80 CP/M system; design of the remaining access structure; implementation of the edit functions to input, delete, and modify data within a relation; and finally the design and implementation of the relational operators. The unique features of the Pascal/MT+ system are described in Appendix A. The access structure will be discussed first since it was an important element in the design of any the functions to add, modify, delete, or retrieve information from a relation.

Access Structure

The access structure used is based on the generalized access structure described by Theo Haerder (1). The access structure consists of the relation file, the B-tree and the leaf structure. The B-tree and its associated structure and operations are discussed in Chapter 2 since they had been previously designed and implemented (5). The relation file and the leaf structure will be described in the following paragraphs.

Relation File

The relation file is the actual repository of information for a specific relation. It contains two types of records: the header record and the tuple record. The header record occurs only once at the beginning of the relation file. It contains information used by the modules that insert tuples into the relation file and information about the number of tuples currently in the file. Figure 7 shows the format of the relation file header. Following the relation file header are the tuples. Figure 8 shows the format of the tuple record. Each tuple contains a deletion field, a type record field, and a list of the attribute values of the tuple. The tuple records are fixed in length but they do span blocks. Each attribute field within the tuple is fixed in length. The length is determined by the domain size definition for that attribute as defined in the data dictionary. The attribute values are stored as ASCII characters; therefore, all numeric fields are converted to ASCII characters before being stored. If the attribute value does not completely fill its field, the remaining characters of the field are filled with blanks.

The relation file has three operations that change it. They are insertion, modification, and deletion. When a tuple is inserted into the relation file it is always inserted at the current end of file as indicated by the end of file pointer in the header. Modification changes the values in the appropriate fields of the tuple and returns the tuple to its original position in the relation file. The deletion algorithm just sets a flag in the deletion

Fields in Relation File Header Record

A	B	Field C			Field D			E	F	G
		C1	C2	C3	D1	D2	D3			
1	1	15	3	3	15	3	3	5	5	5

Length in Characters

Field Defintions:

A - Deletion Field: A field indicating if there are any valid tuples in relation file.

B - Record type: An 'H' in this field indicates a relation file header record. It is to used to verify the correct position.

C - End-of-File Pointer: A pointer to the current last character in the relation file. It consists of three subfields.

C1 - Filename: Filename of relation file.

C2 - Blocknumber: The number of the block in the file where the current end of file is located.

C3 - Charnumber: The number of the position in the block where the next character should be inserted.

D - Overflow Pointer: A pointer to an overflow file

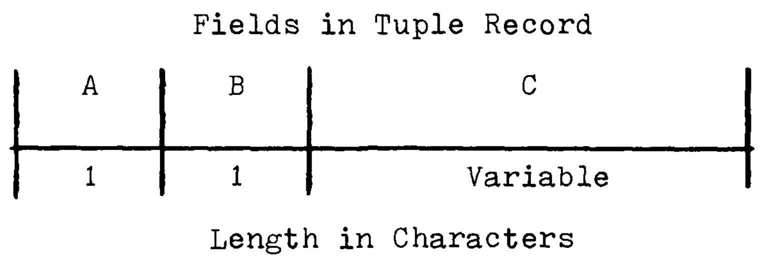
D1 - Overflow Filename
D2 - Overflow Block number
D3 - Overflow Character number

E - Record Size: The size in characters of a tuple. This is the sum of the domain sizes for each attribute in the relation.

F - Number of Tuples: Number of tuples currently in the relation.

G - Deleted Tuples: The number of tuples that have been deleted from the relation file that the space for the records has not been recovered.

Figure 7. Relation File Header Record Format



Field Definitions:

A - Deletion field: A field to indicate if the tuple has been deleted.

B - Record type: A single character to indicate a tuple record. (T)

C - List of attribute values.

Figure 8. Tuple Record Format.

field of the tuple but does not physically remove the data. This does cause some wasted space in the relation file when tuples are deleted. To conserve the wasted space, the insert module could be changed to search the relation file for the first empty space to insert a tuple. Other considerations for future development to better utilize disk space are a reorganization module to reorganize the relation and leaf files to remove wasted space and the use of variable length records to avoid the waste due to blank filling fields.

The remaining structure is the leaf structure. The leaf structure needs to contain all the values for a given domain and pointers to the relations that contain the domain values. Also, the leaf structure should contain enough information so that a given relation can be accessed on the sorted order of a attribute of the relation that is defined in this domain.

Theo Haerder's generalized access structure relies on positional dependencies to depict different characteristics such as owner and member characteristics. Since this is a relational database and it is assumed that good relational database design techniques are employed in designing the relations, it was not considered necessary to keep owner and member characteristics. Therefore, the leaf structure must contain the pointers to the tuples or Tuple Indentifiers (TID) in some form that can be used to access a given relation in sorted order and allow the retrieval of all TIDs to relations that contain an attribute defined on the domain with the same value. The latter property of finding different relations with the same domain value is of great value in simplifying the implementation of the join operator.

The design of the leaf structure also needed to consider the disk storage necessary to store the leaf and how many disk accesses necessary to retrieve a desired TID. Since the B-tree contains an index to a range of values and points to a single leaf node, a structure that placed domain values and TIDs in a single structure would require all leaves to be sequentially accessed until the desired domain value was found. The disk accesses required by this type of structure to reach the desired value is equal to 1/2 the number of domain values stored in a single leaf, assuming that each leaf page contains only one value. If more than one value is stored in a leaf page then the manipulation of the TIDs becomes excessive to maintain the correct orders.

The leaf structure design selected tries to alleviate the problems of disk accesses and simplify the manipulation of the TIDs to maintain a proper order, but does not fully utilize disk space. The decision to use a structure that might waste some disk space but reduce the disk accesses and simplify the necessary manipulation algorithms was made because disk space was plentiful and the optimization of the processing time was one of the goals defined at the start of the development of this relational database system.

Leaf Structure

The leaf structure has two main parts, the leaf header and the leaf pages. The leaf header contains pointers to the previous leaf header and next leaf header and contains domain values and a pointer to the leaf pages where the TID records for each value are stored. Figure 9 shows the structure of the leaf header and Figure 10 shows the detail of the key records. The domain values are stored in a key record that contains the domain value and a pointer to the record in the file that contains the TID records for this value. The key records are maintained in ascending order of domain value within the leaf header. This plus the pointers from one leaf header to the next provide the capability to

Fields in Leaf File Header Record.

A	B	C		D		E		G	H	I
		C1	C2	D1	D2	E1	E2			
1	1	15	3	15	3	15	3	3	3	3

Length in Characters

Field Definitions:

A - Deletion Field: A field to indicate if the whole leaf has been deleted.

B - Record Type: A character to indicate the type of record. 'L' for the leaf header record.

C - Next Leaf Pointer: A pointer to a leaf that contains the records for domain values that are greater than domain values of the current leaf.

C1 - Filename of next leaf.
C2 - Blocknumber of next leaf.

D - Previous Leaf Pointer: A pointer to a leaf that contains the records for domain values less than the domain values of the current leaf.

D1 - Filename of previous leaf.
D2 - Blocknumber of previous leaf.

E - Overflow File Pointer: A pointer to an overflow file. Currently unused.

E1 - Overflow filename.
E2 - Overflow blocknumber.

F - Maximum Number of Keys: The maximum number of domain values (keys) that can fit in the leaf.

G - Number of Keys in Leaf: The number of domain values or keys currently stored in the leaf.

H - Domain Size: The maximum size in characters of the domain values stored in the leaf. If the domain is of type real, one character is added to the domain size to allow for the decimal point.

Figure 9. Leaf File Header Record Format.

Fields in Leaf Key Record

A	B	C	D	
			D1	D2
1	1	Domain Size	15	3

Length in Characters

Field Definitions:

A - Deletion Field: A field indicating if all references of this domain value have been deleted. Currently not used because the record is physically removed.

B - Record Type: A field indicating the type of record. 'K' indicates a leaf key record.

C - Domain Value: The value of at least one attribute that is defined in this domain. The length is the maximum length defined for the domain. If the length of the value is less than the maximum length, the field is filled with blanks to the maximum length.

D - Leaf Page Pointer: A pointer to the leaf page that contains the relation and attribute identifiers and the TIDs of the tuple that contain this domain value.

D1 - Filename of leaf page.
D2 - Blocknumber of the leaf page.

Figure 10. Leaf Key Record Format.

sequentially process a relation on a given attribute. One drawback of this structure is that to determine if a relation's attribute has that value the TID record must be read and searched to see if the relation and that attribute are present.

The physical leaf structure implemented uses the physical considerations that a block is 512 characters in length. Each record has a fixed length since this simplifies insertion, access, and deletion algorithms although it does waste some disk space. The leaf header record contains several pieces of information including the number of domain values that may be contained in a single leaf. The number of values that a leaf may contain is based on the size allowed for the domain. The leaf header record uses 65 characters for its previous and next leaf pointers and other information. This leaves 447 characters of space to be used to store key values and their associated TID record pointers. The key records are 20 characters plus the length of the domain. Since the domain size can range from one to 80 characters, the maximum number of key records that can be stored in a leaf header is 21 and the minimum number of domain values that can be contained in a leaf is 4. By providing a relatively large fanout ratio, the height of the B-tree is kept smaller thus providing for faster processing of the B-tree.

The leaf page records contain three different records -

the leaf page header record, the TID header record, and the TID records themselves. Figure 11 shows an example of a leaf page and how the three type of records fit and are associated in the leaf page.

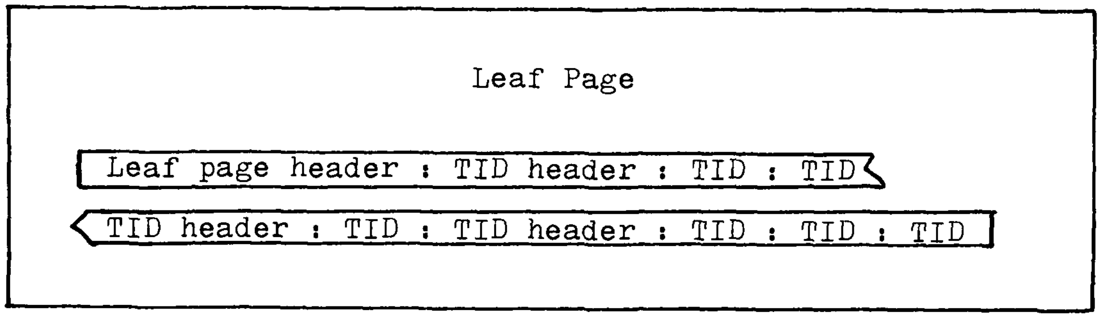

Leaf Page
Leaf page header : TID header : TID : TID
TID header : TID : TID header : TID : TID : TID

Figure 11. Example Format of Leaf Page.

The leaf page header record is found at the beginning of each leaf page record. It provides the number of TID header records and the number of TIDs contained in this leaf page. The leaf page header may also provide a pointer to another leaf page. This pointer is used when there is not enough space in the current leaf page to hold all the TID header records and TIDs for the domain value. Figure 12 shows the format of the leaf page header.

The TID header records are stored based upon an ascending value of the relation filename and the attribute ID. This allows the capability to determine if a relation is present in the leaf page without always having to search the entire leaf page. Figure 13 illustrates the format of a TID header record.

Fields in Leaf Page Header Record.

A	B	C			D	E
		C1	C2	C3		
1	1	15	3	3	3	3

Length in Characters

Field Definitions:

A - Deletion Field: A field to indicate if all the data has been deleted from this leaf page.

B - Record Type: 'H' in this field denotes that this is a leaf page header record.

C - Overflow Pointer: A pointer to a leaf page that contains overflow information from this page.

C1 - Filename of overflow leaf page.
C2 - Blocknumber of overflow page.
C3 - Indicator of which half of the block the overflow is located.

D - Number of TID Records: A count of how many TID records currently exist in this leaf page.

E - Number of TID Header Records: A count of the number of TID header records that are currently contained in this leaf page.

Figure 12. Leaf Page Header Record Format.

Fields in Leaf TID Header Record

A	B	C	D	E
1	1	15	3	

Length in Characters

Field Definitions:

A - Deletion Field: A field indicating if all tuples for the defined relation and attribute have been deleted.

B - Record Type: 'R' in this field indicates that this is a TID header record.

C - Relation ID: A 15 character identifier of a specific relation. Currently the filename of the relation file is used as the identifier for a relation.

D - Attribute ID: An identifier of the specific attribute in the relation.

E - Number of TIDs: A count of TID records that currently exist for this domain value in the defined relation and attribute.

Figure 13. Leaf File TID Header Record Format.

find the correct key record. FINDTID then finds the TID header record in the leaf page that matches the relation and attribute IDs. Then the TID records for that TID header are searched for the TID that matches the TID of the tuple that was deleted. When the correct TID record is found, it is physically removed from the leaf page causing all the TID header records and TIDs to be moved forward in the leaf page to fill the vacant space.

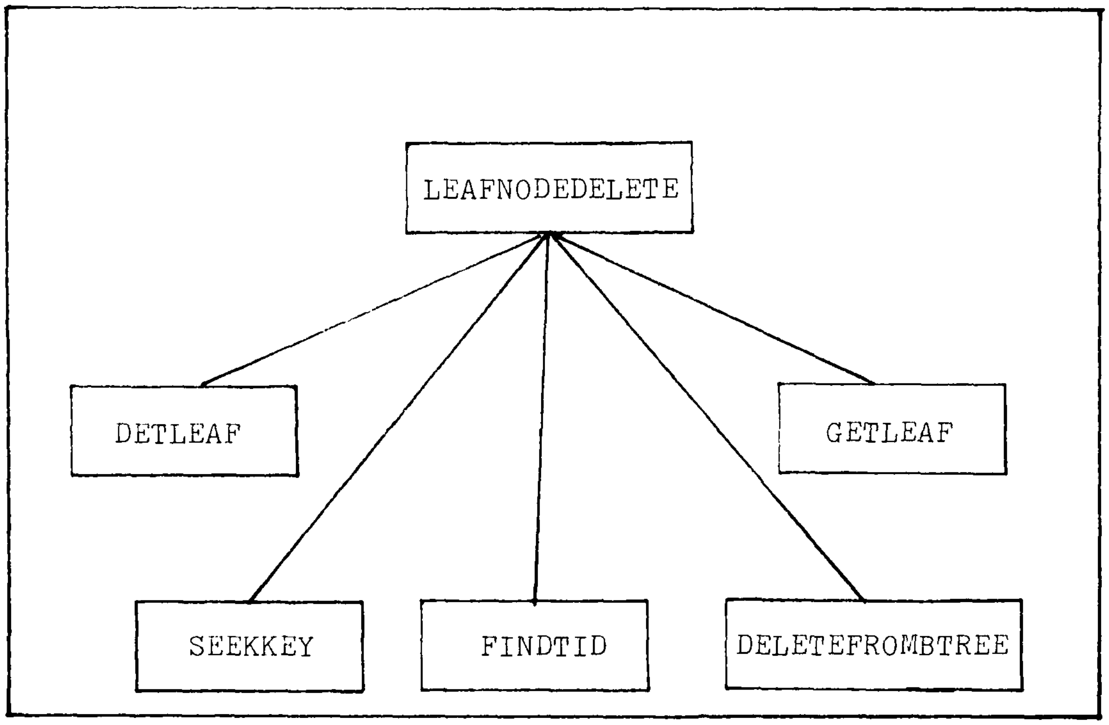


Figure 19. Basic Structure of LEAFNODEDELETE

If the TID record is the only TID for the TID header record, the TID header is also removed from the leaf page. If the leaf page contained only the one TID header record, the leaf page is flagged as being deleted. If the leaf page

The TIDs retrieved are inserted into a list of selected TIDs. This list is an ordered linked list. It is ordered first on relation ID and then attribute ID. This allows the combining of two lists of selected TIDS under "and" or "or" conditions.

GENSELECTLIST takes the selected list of TIDS and calls ORLIST or ANDLIST to combine the list under the condition as stated. If there are conditions for more than one attribute stated, all conditions for each attribute are satisfied and then the list are combined using ANDLIST. This provides a list of TIDs that satisfy all the stated criteria.

DELETE then takes the list of selected TIDs and retrieves the first one. If the user has requested the veto option, the tuple is displayed and the user is asked if he wants to delete this tuple. If the user responds positively, the tuple is deleted from the relation file by placing an '*' in the deletion field of the tuple. The tuple's attribute values are used to delete the tuple's TID from the appropriate leaf file and update the B-tree, if necessary. The DELETE module repeats this for each tuple in the list of selected TIDs.

Leaf Delete Module. The DELETE module calls the LEAFNODEDELETE module to delete the TID from the leaf file and update the B-tree if necessary. Figure 19 shows the basic structure of LEAFNODEDELETE. LEAFNODEDELETE calls DETLEAF to find the correct leaf from which to delete the TID. GETLEAF and SEEKKEY are called to read the leaf and

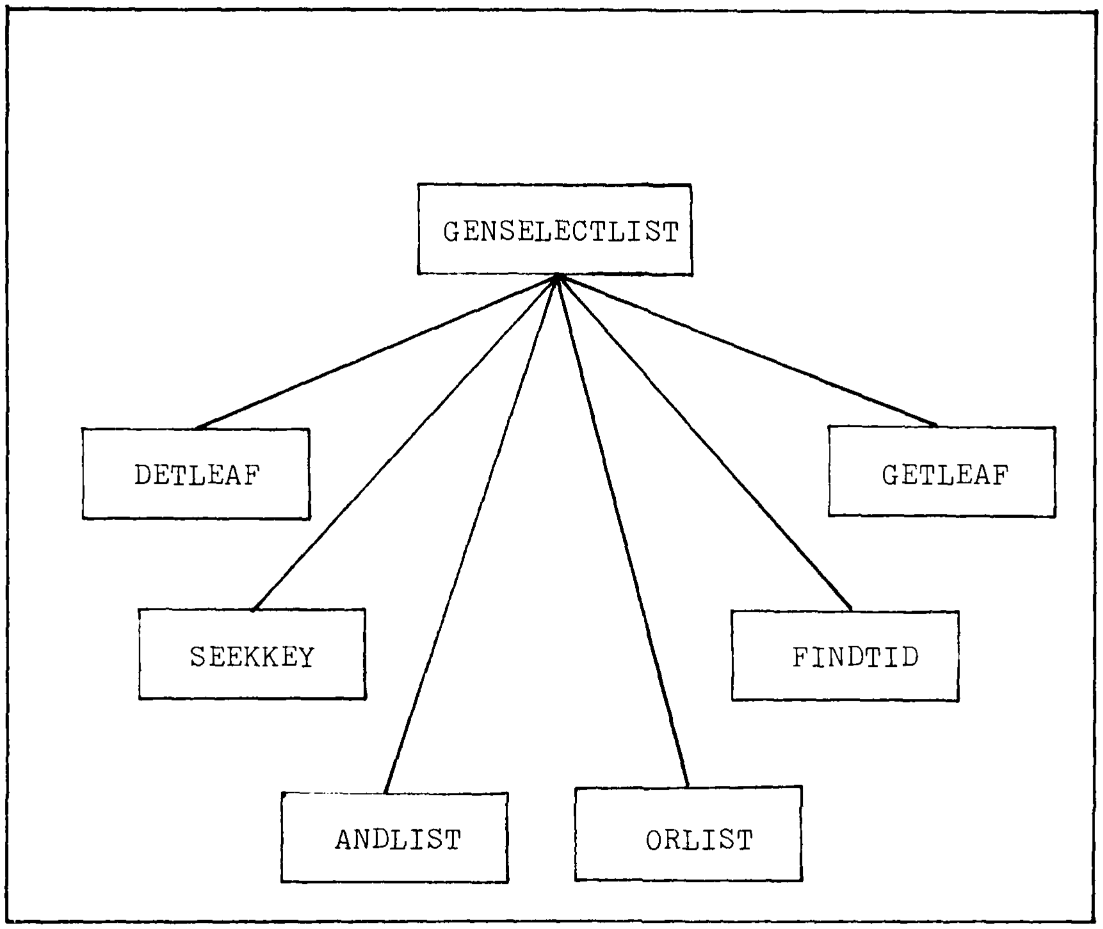


Figure 18. Basic Structure of GENSELECTLIST

key contains the pointer to a leaf page, this pointer is passed to FINDTID which searches the leaf page for a match of the relation and attribute IDs of the attribute value being processed. If a TID header record is found that matches the relation ID and the attribute ID, all of the TIDS associated with that TID header record are retrieved and placed in the select list. The module then proceeds on to the next key record and its leaf page and repeats this process.

to be processed. All attributes of a tuple are kept in the access structure except attributes defined to be of the domain type, text. Therefore, the tuple may be accessed on any attribute value that has been inserted into the access structure which means any attribute not defined as text.

DELETE Module. The DELETE module has three functions to accomplish. First, it must determine all the tuples that meet the selection criteria given by the user. Next, it deletes the tuples from the relation file. And finally, it removes all the attribute values for the deleted tuple from the access structure.

Selecting Tuples Module. GENSELECTLIST is the module that provides the capability to evaluate the selection criteria and then using these criteria produce a list of tuples that satisfy the criteria. Figure 18 shows the basic structure of GENSELECTLIST. GENSELECTLIST works with a single condition at a time. For equal to or greater than conditions it first uses DETLEAF to find the leaf that might contain the given value. If the condition is less than, the module starts processing at the first leaf node which is always maintained in block 0 of the leaf file.

GENSELECTLIST calls GETLEAF to get the necessary leaf. In the case of greater than or equal to, SEEKKEY is then called to find the correct key. If the condition was equal to, only one key is processed. For greater than or less than all keys are processed sequentially until the end of file is reached or until the terminating condition is reached. Each necessary.

Thus the value has been inserted into the leaf structure by LEAFINSERT and its sub-modules. LEAFINSERT returns flags to PUTINLEAFNODE to indicate if the leaf had to be split or if the value inserted was a new maximum value for the leaf. If either of these conditions occurred, then the B-tree needs to be updated. Therefore, PUTINLEAFNODE determines if the B-tree needs to be updated and if so it calls BTREE to update the B-tree.

B-tree Insertion Module. The BTREE module inserts new values into the B-tree. It first inserts the new value into the lowest level B-tree node that points to the leaf. If the leaf was split, then the B-tree is updated by deleting the old value that represented the max value of the leaf and the two values that represent the max values of the two leaves after they were split are inserted. If the leaf was not split but its maximum value changed, then the old value for the leaf is replaced by the new maximum value of the leaf. BTREE then checks the status of the B-tree node after it is updated. If the B-tree node had to be split, then the parent nodes in the B-tree will be updated. BTREE is a recursive module to accomplish the updating of all the appropriate nodes.

After the B-tree has been updated, if necessary, the insertion for this attribute value is complete. The next attribute value of the tuple is then passed to PUTINLEAFNODE largest value currently in the B-tree is returned. PUTINLEAFNODE then calls LEAFINSERT to do the actual insertion of the value.

LEAFINSERT determines if a leaf structure exists for this domain and, if not, initializes a leaf structure. If the leaf structure exists, then the GETLEAF module is called to retrieve the leaf into which the value should be inserted and SEEKKEY determines the position of the appropriate key record or the insertion position of a new key record. If this is a new key, then the leaf is checked to see if it is full. If the leaf is full, then SPLITLEAF is called to split the leaf into two leaves and insert the new key. If the leaf is not full, then the INSERTLEAFKEY module inserts the key record in the appropriate position in the leaf and adjusts the value of the number of keys in the leaf.

If the key was found, then the pointer to the leaf page that contains the TIDS for that value is passed to FINDTID. FINDTID searches for the relation ID and attribute ID of the value being inserted. FINDTID returns the position of the matching TID header record or the position to insert a new TID header record. FINDTID also returns a flag to signal if the matching TID header record was found or a new one needs to be inserted. INSERTTID is then called to insert the TID record and the TID header record, if necessary. INSERTTID has an algorithm that physically moves the TID and TID header records in the leaf page to maintain the proper ordering. It also puts overflow data into new leaf pages as the INSERT module and passed to the PUTINLEAFNODE module.

Leaf Insert Module. The PUTINLEAFNODE module inserts the attribute value into the leaf structure and if necessary calls the module to insert the value in the B-tree. Figure 17 shows the structure of PUTINLEAFNODE.

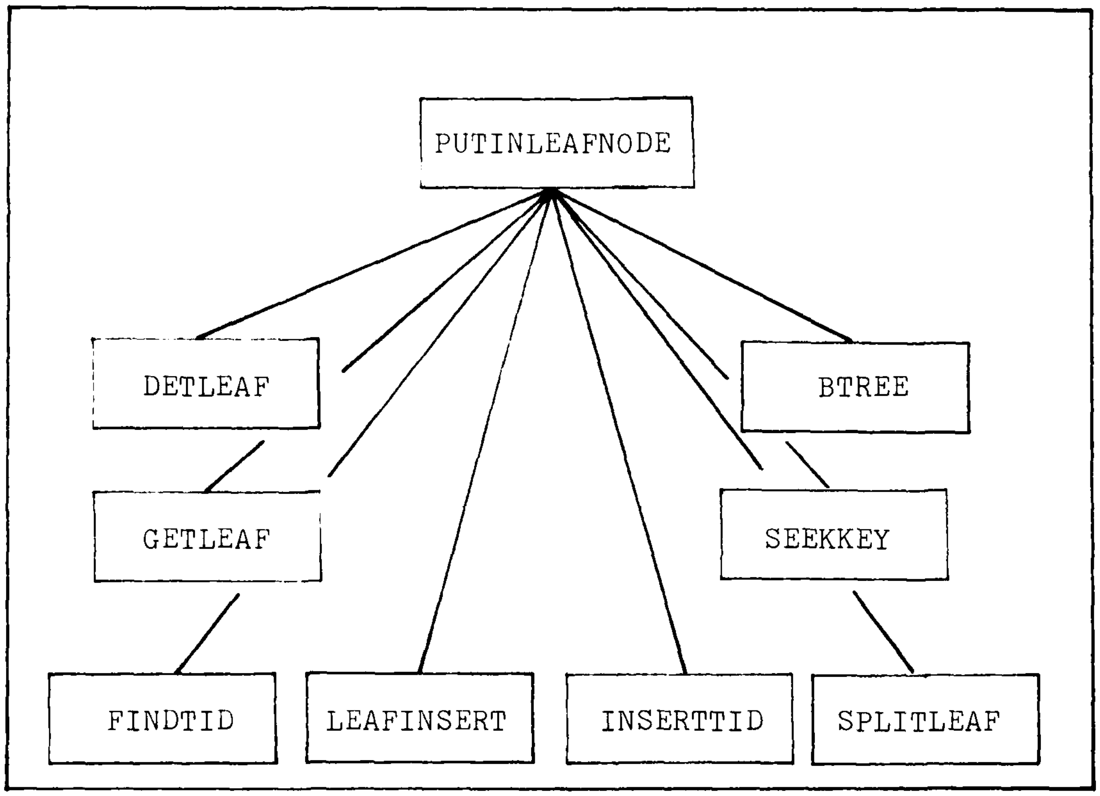


Figure 17. Basic Structure of PUTINLEAFNODE

First, PUTINLEAFNODE calls DETLEAF to get the pointer to the leaf into which the value should be inserted. DETLEAF looks for the B-tree record that has a value equal to or larger than the value to be inserted and returns the leaf pointer of this record. If the value to be inserted is larger than any existing value, then the leaf pointer of the matched but it has not been determined if the value was for the correct attribute of the correct relation so the leaf page is searched for a TID header record that has the exact relation and attribute combination as the key attribute being processed. FINDTID is the module that provides the searching capability of the leaf pages for a specific relation and attribute combination. If a match is found, the key is flagged as being a duplicate.

The duplicate check has to be done for each key of the relation or until a duplicate is not found for a key. This allows the relation to have a key made up of several attributes. If after all the key attributes have been checked and all were duplicates, the tuple is rejected as a duplicate tuple. No further action is taken with this tuple. If the tuple is not a duplicate, the next action is for the tuple to be inserted into the relation file.

<u>Relation File Insert Module</u>. The module that inserts a tuple into the relation is INSERTTUPLE. INSERTTUPLE reads the first block of the relation file (Block 0) and retrieves the relation header. The relation header provides the current end of file for the relation file. The attribute values of the tuple are then inserted at the end of file and a new end of file is placed in the relation header record. INSERTTUPLE returns the TID for the tuple inserted. The TID is actually the starting position of the tuple record that was inserted. The TID is taken by

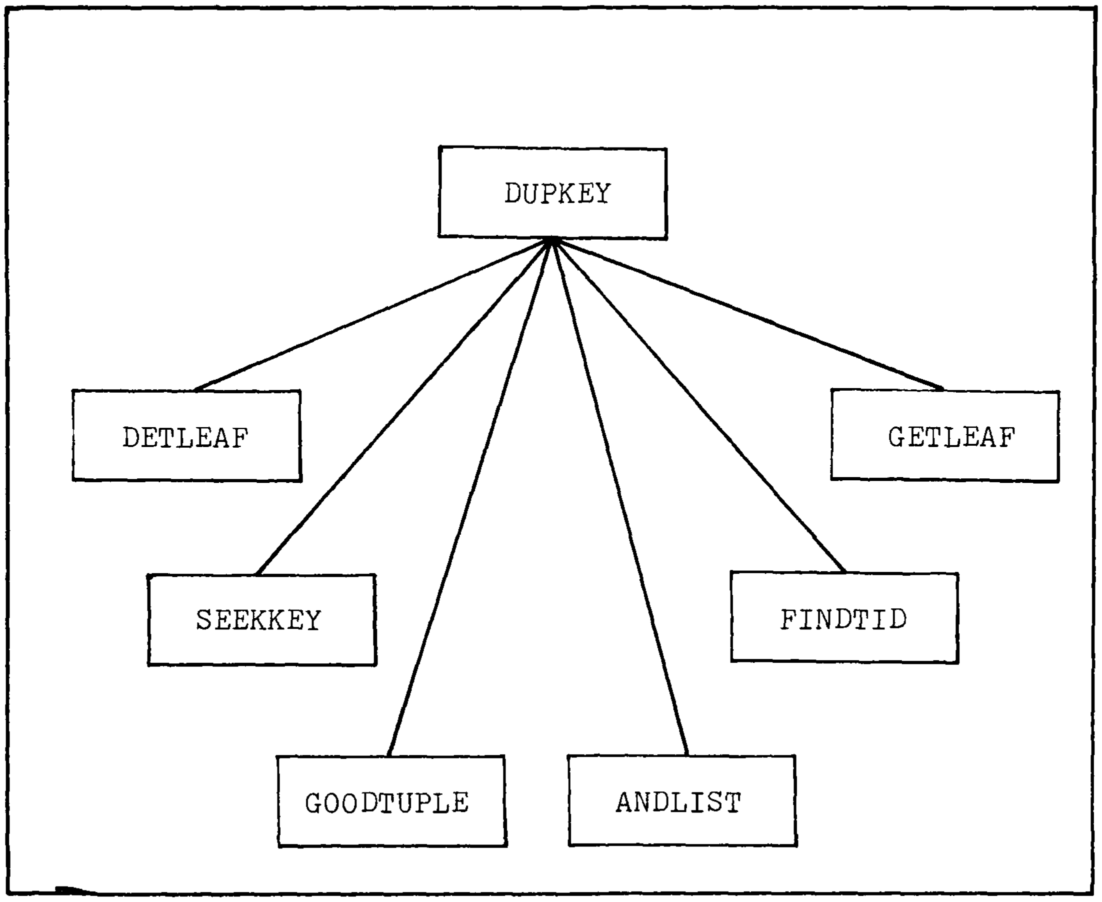


Figure 16. Basic Structure of DUPKEY

takes the pointer and reads that leaf from the leaf structure and places it into the buffer. GETLEAF also retrieves the leaf header record so that the number of keys in the leaf is known.

The leaf's key records are then searched. The module called SEEKKEY does the searching of the leaf key records. If no exact match is made with the attribute value, then this value is not currently defined for this relation or any other relation. If a key record is found with the exact value of the attribute, then the leaf page pointed to in the key record is read into the buffer. The value has now been

The modules are INSERT, MODIFY, and DELETE. These modules will be discussed individually in the following paragraphs.

<u>INSERT Module</u>. INSERT is provided a list of the tuples to be inserted into the relation. It handles each tuple individually to insure that no duplicates might try to be inserted. The insert module has four tasks to perform for each tuple. These tasks are: check to see if the tuple is a duplicate; insert the tuple into the relation file; insert the attribute values of the tuple into the leaf structure; and if necessary, insert the attribute values into the B-tree. First the duplicate checking function will be discussed.

<u>Duplicate Tuple Checking Module</u>. The actual name of the module that checks for duplicates is DUPKEY. Figure 16 shows the basic structure of DUPKEY. DUPKEY references the relation definition stored in the data dictionary for the key attributes of the relation. Then the values of the key attributes of the tuple to be inserted are checked against the current values in the relation. This check is done by taking the value of the first attribute and calling DETLEAF. DETLEAF determines if the B-tree structure exists. If the B-tree does exist, it searches the B-tree for the first value equal to or larger than the attribute value from the tuple. Once the appropriate B-tree record is found that points to a leaf, the pointer to the leaf is returned to DUPKEY. DUPKEY then calls a module called GETLEAF that

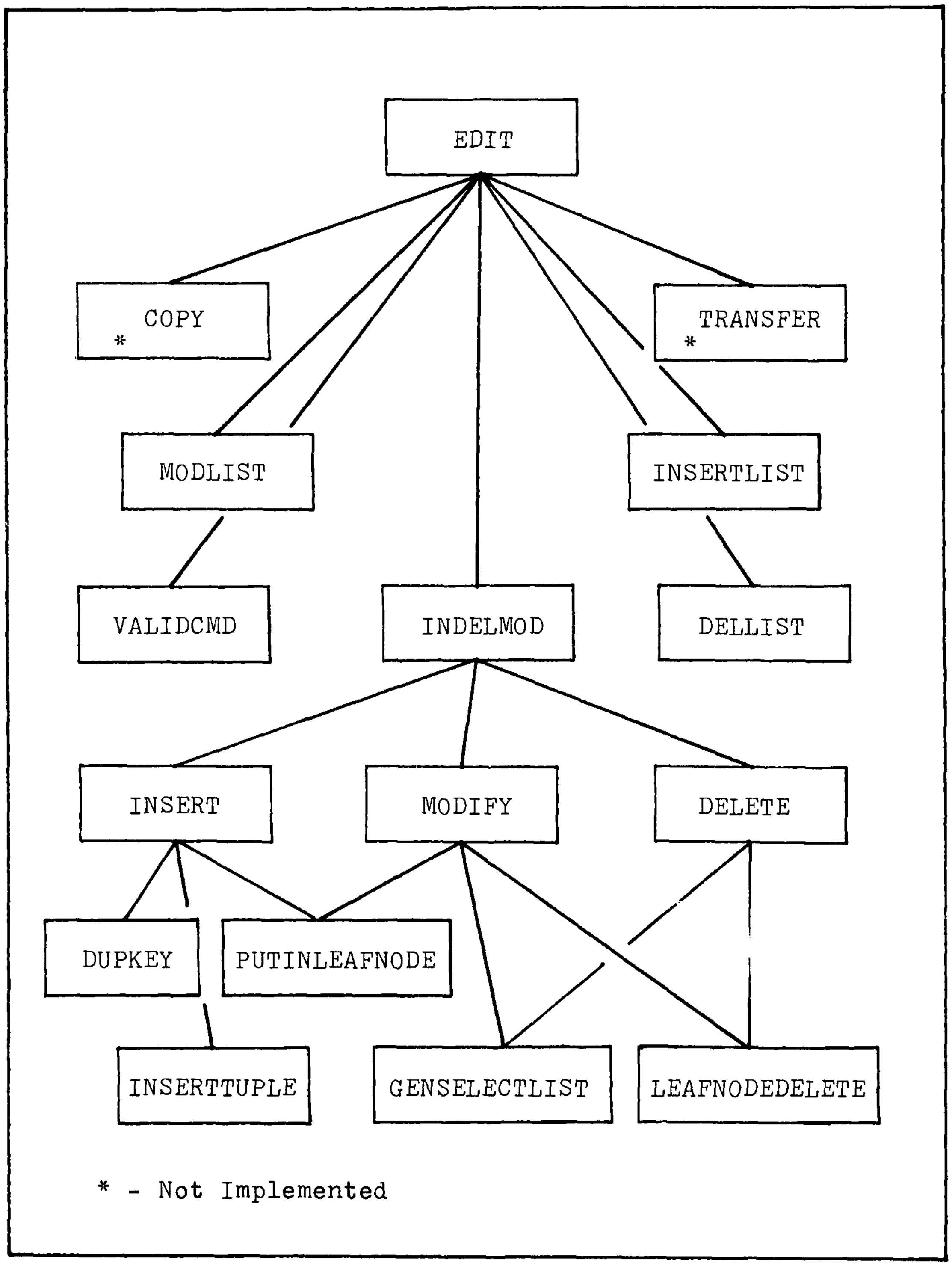


Figure 15. Basic Structure of Edit Module

Basic Edit Modules

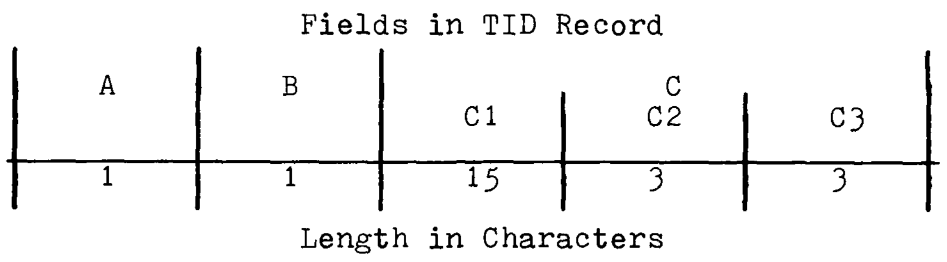
The basic edit modules provide the capability to input new data into a relation, modify existing data in a relation, and delete existing data from a relation. Figure 15 shows the basic structure of the edit module. The edit module starts by requesting the name of the relation to be used. This relation name is verified as a defined relation and the user's ID is compared to see if it matches the owner ID. If the user's ID is not the same as the owner ID, the user is queried for the appropriate ID. If the user cannot provide the correct ID, the process is terminated. This provides a measure of security to the system by only allowing users that are owners or who know the correct password.

Now that the user has provided the appropriate password, the user is asked to input his data. For the insertion routine, the user enters complete new tuples. The modify module requests a set of selection criteria that will identify the intended objects and asks for the values of individual attributes which the user wants to modify. The delete function asks for a set of selection criteria for the tuples the user wants to delete. Both the delete and modify functions provide a veto capability to allow the user to verify that he wants to change the tuples selected. After the command and data is verified with the user, INDELMOD is called. The INDELMOD module determines which type of command is being processed and calls the appropriate module.

The records in the leaf page are all fixed length records. This allows the algorithms that search the leaf page to skip over TIDs that do not meet the criteria being searched for. Also, this allows the insertion routines to maintain a physical ordering of the TID header records by moving a fixed number of characters to make room for an insertion. It should be noted that some fields are not always fully utilized so these fields are blank filled to provide the proper length. One such case is the filename field of the pointer fields since it is defined as 15 characters but only 6 characters are currently used. The extra space was allowed to provide expansion room in case the system was installed on a machine that allowed more characters for a filename or to provide for the inclusion of some suffix to the filename to provide for different versions or backup files with the same name.

The leaf structure was designed to be an independent data structure. This will allow the leaf structure to be changed independently of the B-tree structure or any of the other functions of the system. To provide this independent data structure several modules were developed that provide the capability to find a domain value in a leaf, find the location of a TID for a given relation and attribute value, insert a new domain value in the leaf, insert a TID into the leaf structure, and delete a TID from the structure. These modules will be described in conjunction with the basic edit functions that insert, delete, and modify relations.

Following every TID header record is at least one TID record. If an attribute in a relation has the same value in different tuples there will be a TID record for each tuple, containing that value, following the TID header record. The TID record actually contains the pointer to tuple in the relation file shown in the TID header record. Figure 14 shows the details of a TID record.



Field Definitions:

A - Deletion Field: A field used to indicate if the TID has been deleted.

B - Record Type: 'T' in this field indicates that this is a TID record.

C - TID: This is the tuple identifier. It points to the file, block, and character position where the tuple can be found.

C1 - Filename of the relation where tuple is located.
C2 - Blocknumber of the relation file where the start of the tuple is stored.
C3 - Character number in the block where the first character of the tuple record can be found.

Figure 14. TID Record Format.

was an overflow page, the necessary adjustments are made to any preceding leaf pages that referenced it and any leaf pages that might have followed it. If the leaf page was deleted and it was the only leaf page for the key record, the key record is then deleted from the leaf.

The leaf key record is physically removed from the leaf. This causes all of the key records that followed it in the leaf to be shifted forward one position. This maintains the ability to sequentially access a relation that has an attribute defined on this domain. If the key record that was deleted was the maximum key record of the leaf, then DELETEFROMBTREE is called to adjust the B-tree.

B-tree Delete Module. DELETEFROMBTREE is the module that updates the B-tree as necessary. If the maximum value of a leaf is changed, then the value for the leaf is changed in the B-tree. If a leaf was completely emptied by the deletion, the reference to the leaf is deleted in the B-tree. The B-tree then recursively adjusts any parent nodes as required. This may mean collapsing the B-tree if a B-tree node is completely emptied.

These operations are performed for each attribute in each tuple that is not defined as domain type, text. The DELETE module selected a complete list of TIDs first and fully processes the tuple for each TID before going to the next TID.

MODIFY Module. The MODIFY module acts like a delete and modify module but only for selected attributes within a tuple. First, MODIFY calls GENSELECTLIST to provide a list of TIDs. Then each tuple is retrieved from the relation list and the user has the option to have the tuple modified or leave the tuple as is. If the tuple is selected to be modified, the attributes to be modified are changed in the tuple and then it is placed back in the relation file. Then each attribute that was modified is first deleted from the leaf file by LEAFNODEDELETE and the new value for the attribute is inserted into the leaf file by PUTINLEAFNODE. The descriptions of both of these modules is provided above. This method of modifying the tuples does have one drawback. It does not guarantee the integrity of the database because when an attribute is modified it is not checked to see if it is part of the key for the relation or if the value being inserted is a duplicate value.

Implementation of Relational Operators

The relational operators all are included in the part of the relational database system called the RUN module. The RETRIEVE module consists of the submodules to: build and edit a query file; perform a syntax check of the query file and produce a query tree(s); optimize the query tree; retrieve the data as requested by the queries; and display a relation. The modules to build, edit, check, and optimize were previously implemented (6). Figure 20 shows the basic structure of the RETRIEVE module. Therefore, the following

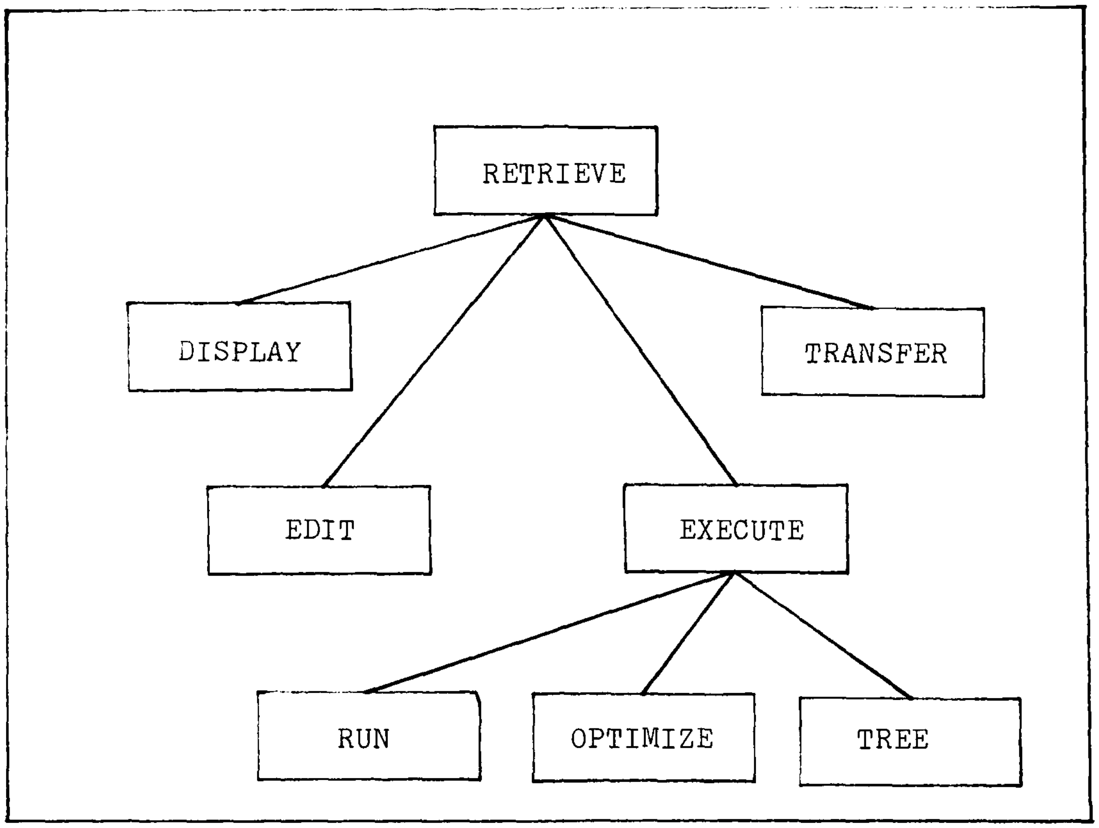


Figure 20. Basic Structure of RETRIEVE Module

discussion focuses primarily on the implementation of the relational operators (PROJECT, JOIN, SELECT) and the module to display relations (DISPLAY). It should be noted that due to time limitations not all of the relational operators have been implemented. The relational operators implemented are those used most frequently in queries. It was felt that these should be implemented first.

The first step of the RUN module originally was to take the query tree and perform the operation called the coordinating operator constructor (7). The coordinating operator constructor was previously implemented (4) but was never fully tested. The coordinating operator constructor attempts to put intermediate relations into a preferred sort order based on attributes that will be used in further queries. Since the access structure implemented provides a directory for every attribute, the function of the coordinating operator constructor was not necessary. Therefore, the coordinating operator constructor was not used as the first step of the RUN module. Thus, RUN gets the bottom query from the optimized query tree and calls the appropriate relational operator to perform the query. The procedure is repeated for each query moving upward in the query tree. When the root of the tree has been processed, the RUN module returns control to the EXECUTE module. The EXECUTE module may then have the RUN module process another query tree or return control to the RETRIEVE module. Figure 21 shows the basic structure of the RUN module.

All of the relational operators share a common first step. The first step for each relational operator is define the resulting relation of the query in the data dictionary. The DEFTEMPREL module provides the procedures necessary to complete the data dictionary definition of the resulting relation.

DEFTEMPREL accesses the data dictionary and retrieves the definition for the source relation. This becomes the basis for the definition of the new resulting relation. The resulting relation is a temporary relation so it is assigned

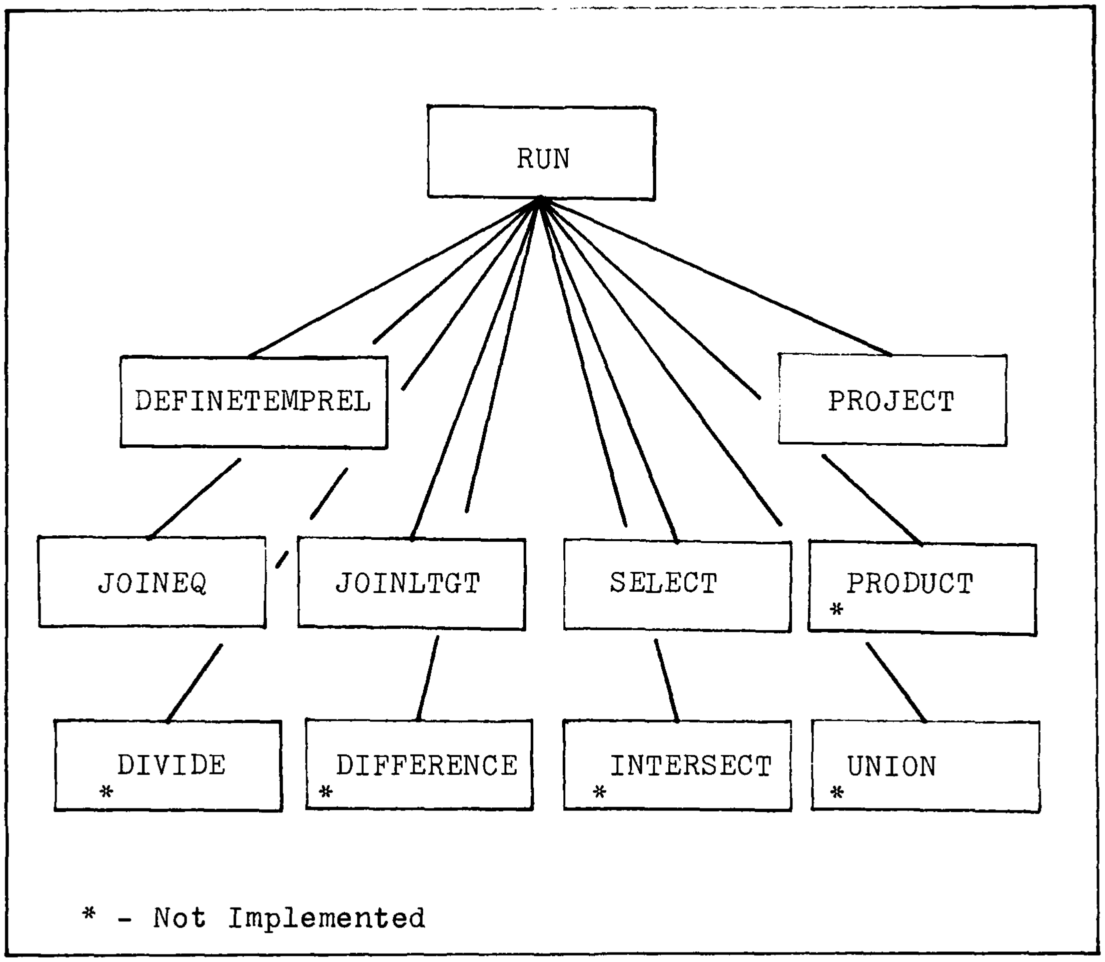


Figure 21. Basic Structure of RUN Module

a relation ID of the form T followed by five digits. The largest temporary relation identifier is stored in a global variable and incremented each time a temporary relation is defined. The global variable is reset each time the program is initiated. The DEFTEMPREL module next needs to define the attributes in the relation but this differs for the different relational operators so this will discussed under each individual operator. First, the DISPLAY module will be discussed followed by the relational operators.

DISPLAY Module. The DISPLAY module was designed to allow the user to display a relation. It allows the user two options of how to display the relation. Figure 22 shows the structure of DISPLAY. The first option is to display the relation in the order it was inserted into the relation file. This method provides no order except a chronological order which is generally not important since the records in the relation are not time-dated. The other option is to display the relation sorted on one of the attributes in the relation. The design, the advantages, and the disadvantages of each of the options will be discussed in the following paragraphs.

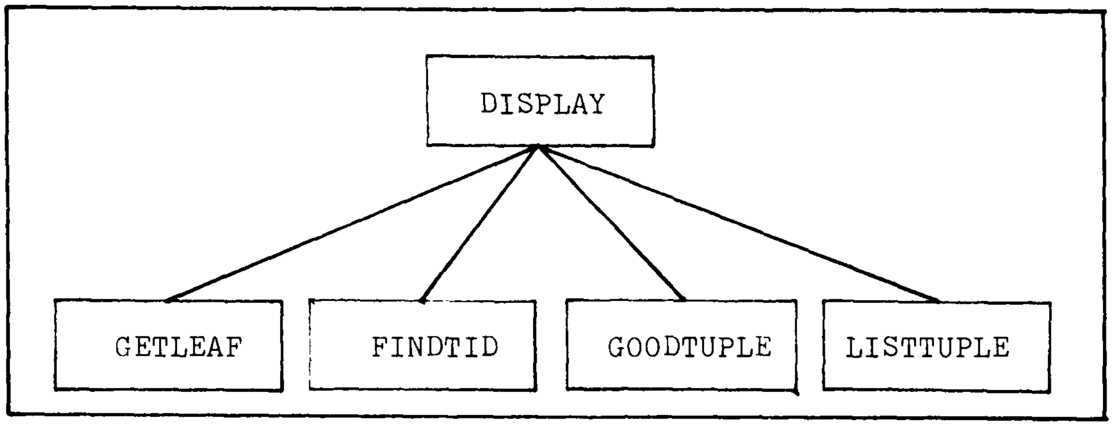


Figure 22. Basic Structure of DISPLAY

The option to display the relation in insert order has very little design involved. It simply goes to the relation file and reads the relation header. This provides the size of the tuples. Then each tuple is read and displayed on the screen. The display on the screen simply puts the value of the first attribute on the screen and follows the value with a semicolon and a space and then the next attribute value is displayed on the screen. This method of displaying the relation is somewhat crude but it is simple and very efficient. Since no indices are ever read to display the relation, this option of displaying the relation is very fast.

The sorted option is much nicer to read since it displays the relation sorted on an attribute of the user's choice. But the sorted option is somewhat slower since it must read an index and then retrieve a tuple. The design of the sorted option used many of the existing sub-modules developed during the development of the edit functions to search the access structure, which provides the index.

The first step in the sorted display after the relation name is verified is to request the attribute name, from the user, with which the relation should be accessed. This attribute name can be any attribute in the relation that is not defined to be the domain type, text. The domain type text does not have an access structure so it may not be used. This is noted here because this will cause some limitations in implementing the relational operators. The attribute name is then verified to be an acceptable attribute name.

The retrieval of the relation is now ready to begin. The first step is to read the first leaf in the leaf file for the domain under which the attribute is defined. The first leaf is always block 0 of the leaf file. GETLEAF is used to read the leaf. Next, the first key record is read used to read the leaf. Next, the first key record is read from the leaf and the leaf page pointer given to FINDTID. FINDTID then either finds the TID header record for the relation ID and attribute ID of the chosen relation and attribute or returns a flag showing that the relation-attribute pair is not present for this domain value. If the TID header was found, all of the TIDs that belong to this TID header are read. The tuples defined by the TIDs are then retrieved from the relation file and displayed.

After all of the TIDs for the key have been processed, the next key value is read from the leaf and the proce[illegible] repeated. When all of the key records in a leaf have been read, the next leaf that is pointed to in the leaf header is read and its keys processed. This continues until all of the leaf file has been processed.

The sorted option of displaying a relation has some shortcomings. The first is the ability to display the relation only in ascending order of the attribute. This is caused only by the fact that the pointer to the last leaf or leaf with the largest value is not stored. If this pointer were stored somewhere, i.e. the leaf header of the first leaf, then this capability could exist because each leaf header has the pointer to the previous leaf.

The second shortcoming is the fact that since each leaf page has to be accessed to determine if a TID header is present for the desired relation, the retrieval is very slow. At the present time there is no easy method to improve this since it would require a different access structure to alleviate this problem. Since displaying the relation is very closely related to the operations necessary to implement the relational operator, PROJECT, it will be discussed next.

PROJECT. The relational operator PROJECT, as all of the relational operators, must first define a resulting relation in which to store the results. The DEFTEMPREL module provides the procedures to define the resulting relation in the data dictionary. After DEFTEMPREL defines the relation, it defines the attributes. The attributes defined in the PROJECT operator are the only ones defined and they are defined in the order that they are found in the project operator. Thus the project operator, PROJECT rel1 OVER att3, att2 GIVING temprel1;, would have a different data dictionary definition than the operator, PROJECT rel1 OVER att2, att3 GIVING temprel1;.

The next step in the PROJECT operation is to retrieve the original relation. This is done exactly like the DISPLAY module retrieves the relation in the unsorted option. As each tuple is read from the relation file, its attribute values are stored in a record with the attribute names. Thus to complete the project for this tuple, the data dictionary definition of the resulting relation is used to select the attribute names and values from the tuple. Each selected attribute value is then placed in a tuple of the resulting relation. The new tuple is now ready to be inserted into the resulting relation. First, the new tuple must be checked to see if it is a duplicate tuple. This will require the DUPKEY module to be used to check if all the values in this projected tuple have already been placed in the resulting relation file. If the tuple is a duplicate, it is discarded and the next tuple retrieved from the original file. If the new tuple is ok to be inserted into the resulting relation file, then the INSERT module of the edit function is used to insert the tuple and its attribute values into the access structure. This process is repeated for each tuple of the original file. When the process is complete, the new resulting relation can be displayed or used as any other relation in queries. The resulting relation and its associated data will remain in the data dictionary and access structure until the user decides to quit working and terminates the program. At that time the temporary relations will be deleted from the data dictionary, the TID header and TIDs will be removed from the access structure, and the temporary relation file deleted. Figure 23 shows the structure of PROJECT.

SELECT Module. The SELECT operator starts by calling the DEFTEMPREL module to define the resulting relation. The attribute definitions for the resulting relation are exactly the same as the source relation named in the SELECT command. The next step is to select the appropriate tuples from the source relation.

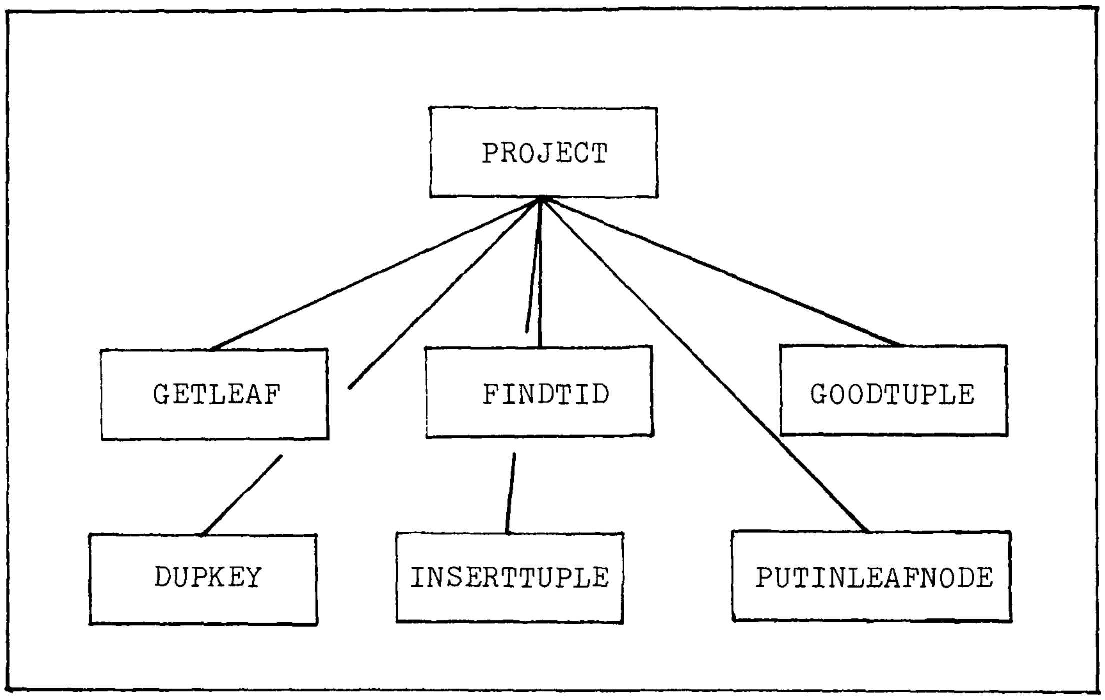


Figure 23. Basic Structure of PROJECT

The criteria for the select operation are read from the query tree. Each criterion is passed to GENSELECTLIST which produces a list of TIDs that satisfy that criterion. The list is then combined with the previous list, if there is one, using the operator in the query - either ANDLIST or ORLIST as is appropriate. The syntax checker, the TREE module of the RUN module, removes the parenthesis from the query if any were present and orders the criteria into the correct evaluation order. The list of selected TIDs contains no duplicates because the TIDs are inserted into the select list in an ascending order of the relation ID - attribute ID combination and duplicate TIDs are eliminated. When the ANDLIST module is called to combine two lists of

TIDs, it eliminates all TIDs that are not found in both lists. The ORLIST module combines the two lists using all of the TIDs of both lists but eliminates the duplicates if a TID appears in both lists. Therefore, when all the criteria have been used, the resulting list is the final list of TIDs that satisfy all of the stated conditions.

Next, the tuples identified by the TID list are read from the relation file. As each tuple is read from the relation file it is inserted into the resulting relation file. Then the attribute values of the tuple in combination with the TID of the tuple are inserted into the access structure. The tuples inserted into the relation file are not checked to see if they are duplicates because it is assumed that the source relation had no duplicate tuples and SELECT does not introduce any new tuples or modify the attribute values of any tuple to cause a duplicate.

The SELECT operation is complete once all of the TIDs in the selected list of TIDs is processed. The resulting relation is now available for all operations. The resulting relation will remain until the program is terminated at which time all traces of the resulting file will be deleted. Figure 24 shows the basic structure of SELECT.

JOIN Module. The JOIN relational operator is actually three different operators: JOIN>, JOIN=, and JOIN<. The three operators in one all share the same procedure of defining the resulting relation, selecting the tuples from the two source relations, and finally inserting the

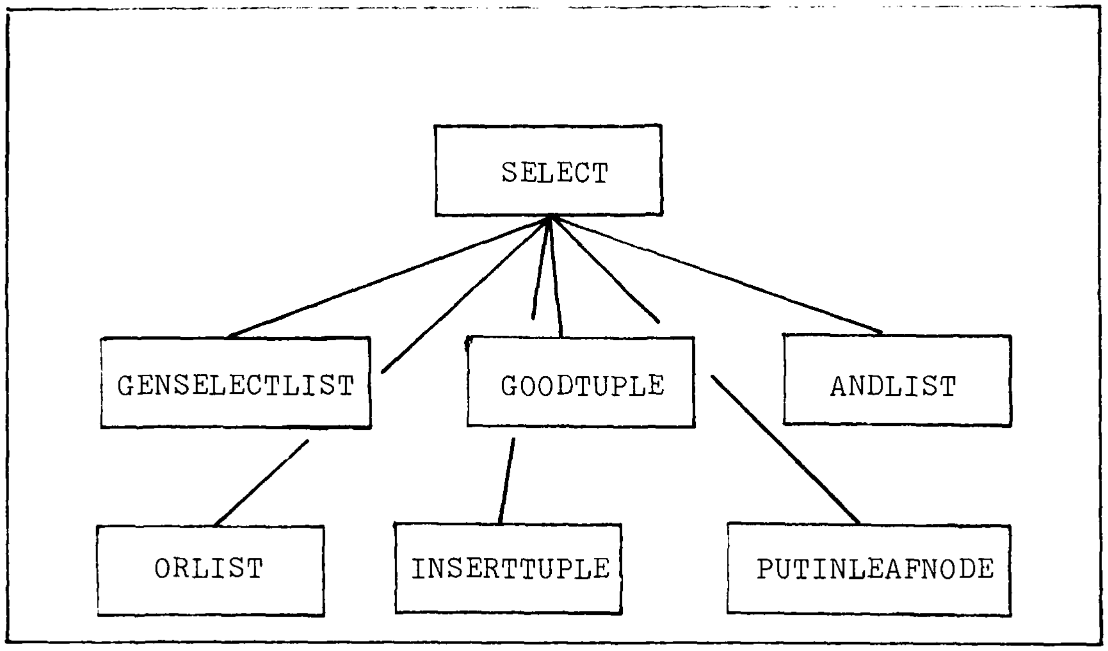


Figure 24. Basic Structure of SELECT

resulting tuples into the relation file and the tuples' attribute values into the access structure.

First, each of the different joins defines the resulting relation by calling DEFTEMPREL. DEFTEMPREL defines the resulting relation by combining the attribute definitions cf both of the source relations. If duplicate names exist for the attributes in the source relations, the name of the relation is appended as a prefix to the duplicate names. If the names are still the same, i.e. joining a relation to itself, suffixes of -1 and -2 are added to the attribute names to make them unique.

The resulting relation is now defined in the data dictionary. Next, the appropriate tuples from each of the source relations need to be selected and joined together to form the tuple of the resulting relation. The method of selecting the appropriate tuples will be discussed later for each individual join, but first the insertion process will be described.

After the join determines a pair of TIDs that meet the criteria provided, the selected tuple from each source file must be read and the two tuples combined to make a new tuple of the resulting relation. The TID of the first source file will be used first to access the tuple in the relation file. This tuple's attribute values can then be placed in the new result tuple. Next, the tuple from the second file is read and its attribute values placed in the resulting tuple. This creates a tuple that is the combination of the two source tuples. This tuple can now be inserted into the resulting relation file. After the tuple is inserted into the resulting relation file, each of the attribute values in the resulting tuple is inserted into the access structure so the resulting file could be used as a source file in later queries.

No check for duplicates tuples is made because the design of the joins does not introduce duplicate tuples. The design of the join operators uses the fact that each of the tuples in the two input relations is unique (if the input relations can contain duplicate tuples a check for duplicate tuples would be necessary). Then the join function uses one of the input relations as the control. This another purpose. The value of the BAR attribute was maintained as the same value in each tuple, causing a leaf page to overflow to test the ability of both finding values in a overflow leaf page and also causing the algorithm that makes room for the inserts in the leaf page to move data to an overflow page.

The last special case to be tested in the INSERT module was recognizing a duplicate tuple. So the tuple:

```
DRINKER = TOM;
BAR = 45;
BEER = 45.9;
```

was again input into the INSERT module. This tuple was rejected as a duplicate tuple since its key values were already defined in the access structure for this relation. The test of only one key being the same had previously been done because of the values inserted for the BAR attribute. Finally, the DRINKER_DATA relation had two tuples inserted that had the same value for DRINKER that had been inserted earlier with the FREQUENTS relation. This tested the ability of the INSERTTID module to correctly insert new TID header records in a leaf page that already existed. This verified the INSERT module and provided data to be used for testing the DELETE and MODIFY modules.

DELETE MODULE. The DELETE module has three primary tasks to accomplish. The first task involves verifying the relation name provided by the user, verifying the user is authorized to do deletions, and building the selection criteria. The next task is to use the B-tree and leaf files in value in a leaf and also testing to be sure the B-tree module was invoked to change the value it contained for this leaf. The BAR attribute tested the ability of the procedure LEAFFIND function to find a defined value and recognize it as a previously defined value. This also tested the ability of the INSERTTID module to correctly add the TID behind a previously created TID header and update all appropriate fields. The BEER attribute tested the ability to insert a key value in the correct position in the leaf by moving the other key records to make space for the insertion.

The INSERT module was provided with four more tuples to be inserted. These tuples caused the first leaf of the leaf file for the domain for the DRINKER attribute to be filled. Thus the next tuple inserted caused the first leaf to be split. This tested the ability to correctly split the key values and update the appropriate previous leaf and next leaf fields in the leaf headers. It also tested the BTREE module to insure its ability to update the B-tree with the values for the new and old leaf.

The test was continued until a second leaf was caused to split to insure that the leaf pointers that maintain the sequential ordering were correctly maintained. One test that was not completed was testing the BTREE module to insure that it correctly split its nodes. This was not tested because of lack of time and the fact that this capability had been tested previously.

The tuples inserted to cause the leaf to split also had

```
DRINKER = TOM;
BAR = 45;
BEER = 45.9;
```

Now, the real test was ready to begin. The first tuple inserted tested the ability of the module to recognize an empty relation file and initialize the relation file before inserting the tuple. The relation file was examined to insure the file was initialized correctly and the tuple correctly inserted. Next, since no B-trees existed both leaf files and B-trees had to be initialized for the appropriate domains. Then the first attribute values were inserted into the leaf and this caused a leaf header record to be created and a key record inserted. Then the first leaf page was initialized and a TID header record created followed by a TID record. Then the BTREE module was called to initialize the B-tree and insert the first value and the leaf pointer. The disk files were examined to insure that all of the records in all of the files were correctly initialized and placed in the files.

The next tuple to be inserted had the following attribute values:

```
DRINKER = ZURD;
BAR = 45;
BEER = 45.3;
```

Each attribute value was designed to test a different phase of the insert algorithm. The DRINKER attribute tested a boundary condition because it became the largest value in the leaf thus testing the ability to insert the greatest leaf files, and the B-trees. The procedures that handle the relation files and the leaf files were more thoroughly tested than the B-tree functions since the B-tree functions were supposedly verified during their development (5: 96-109). Obviously, the insert algorithms had to be tested and correctly working before the other functions could be tested. Therefore, the INSERT module testing will be described first.

INSERT Module. The first step in testing the INSERT module was testing the algorithms that verify if the user has the correct user ID or knows the correct insert security ID. This was a self checking type of function because if a valid answer was provided and the function did not recognize it, then the system stopped processing the command.

The next step of the test was to query the user for a relation name to be used for the insert. Again this was self checking because if the function did not recognize the relation name as valid, an error message appeared. The procedure was also tested to be sure an error message appeared and processing ceased when a known undefined or bad relation name was given. Once the relation name is validated, the system requests the attribute values of the tuple to be inserted. The values are printed back out on the screen after entered by the user to verify they were translated and received correctly by the system. The first tuple to be inserted was for the relation FREQUENTS and had the following values:

These relation definitions were selected because they provided attributes of every domain type and a different amount of keys. These characteristics are important considerations in further testing.

The verification of the definitions of the data definition module was done by examining the definitions as written on the disk file called SETUP.DAT. Since this module had just been converted, the format of a correct definition was known and was compared to the existing definitions. One important element of the definition that was checked was the system supplied relation ID. The verification of this was done by first defining the relation, FREQUENTS, and saving the definition on disk. The program was then rerun and the definition of the relation, DRINKER_DATA, was added and the definitions stored. The second definition did have the proper ID of R00002. This was significant because of the fact that it proved that the algorithm used to determine the greatest relation ID previously used functioned correctly.

The verification of the data definition module provided some domain and relation definitions. These definitions will be used for the testing of the next module, the Edit module.

Edit Module Testing

The verification of the Edit module involved testing the INSERT, DELETE, and MODIFY functions. These functions involve manipulating the data in the relation files, the modules could begin.

The testing of the data definition module consisted of defining five different domains and two relations. The domain definition consists of the name of the domain, the domain type, and the maximum size in characters. The test domains were:

DRINKER, Char, 30 characters;
BAR, Integer, 10 characters;
BEER, Real Number, 5 characters before the decimal point and 5 characters behind the decimal point;
ADDRESS, Char, 20 characters;
COMMENT, Text, 30 characters.

The domains were defined and then two relations were defined using these domains. The sort orders of the attributes in the relation are not used so the definition of the sort orders was not thoroughly tested. The relation's definitions consisted of the name of the relation, the name of the attributes (the domain of the attribute will be shown in parenthesis), any constraints of the attribute value, the owner ID for the relation, and the security IDs for reading, inserting, deleting, and modifying. Also, the attributes that form the key for the relation were defined. The following are the relation definitions used:

Relation Name = FREQUENTS; Attributes = DRINKER (DRINKER), BAR (BAR), BEER (BEER); Constraints = BAR must be $<$ 100; Owner ID = TGK; No security IDs were defined; Key = DRINKER, BAR.

Relation Name = DRINKER_DATA; Attributes = DRINKER (DRINKER), ADDRESS (ADDRESS), COMMENT (COMMENT); No Constraints; Owner ID = TGK; Read ID = READ; Insert ID = INSERT; Delete ID = DELETE; Modify ID = MODIFY; Key = DRINKER.

IV. Verification and Validation

Verification and validation are important steps in the development of a system. Verification is the process of testing the system to ensure each set in the processing produces the expected response. Validation ensures that the system functions meet the prescribed requirements. Primarily, verification of the system will be described because of the lack of true requirements to be validated.

The verification of functions can be done in different ways. The method of testing every case or exhaustive testing was not feasible, so the test cases selected were selected either to provide a boundary value analysis or to ensure that every logic path was executed at least once. The functions being tested and the test cases will be described in the following paragraphs.

Data Definition Testing

The first task of the system development was to convert the existing code from UCSD Pascal to Pascal/MT+. Successful compilation was the primary verification used at this time. The converted code was more completely tested during the verification of the functions that were developed during this thesis effort. The one exception that was verified at this time was the data definition facility. This module defines the domains and relation. These definitions had to be present in the data dictionary before testing of other in this chapter. Next, the procedures used to test the access structure modules and relational operators will be discussed.

very similar to the JOIN> operator. The only difference is that the processing of the second relation starts at the next larger key than the first relation and continues through all the remaining larger keys instead of starting at the smallest key value and continuing to the value of the first relation. This process does not provide any duplicate tuples, thus allowing the insert portion of the algorithm to not check for duplicate tuples.

PRODUCT. The PRODUCT module is not currently implemented, but the design of algorithm is obvious from the JOIN operators. To provide a product, the first relation would be processed once for each key and the second relation processed for all the key values for each key of the first relation. Once again, this method avoids the problem of having to check for duplicate tuples because none are introduced.

Summary

The implementation of the relational database consisted of the following main tasks: converting the existing code; designing and implementing the access structure and the modules to manipulate the access structure; and designing and implementing the algorithms to provide the relational operators. The details of converting the existing code and the problems that arose during further development of the system are documented in Appendix A. The implementation of the access structure and relational operators were described finding the TID header of the first relation. As long as the second value is less than the retained value of the first, the processing continues. If the second value is less than the first value, then FINDTID is called to find the TID header of the second relation. When a TID header is found for the second relation, then each TID of the second relation is combined with each TID of the first to form TID pairs. These pairs can now be used to access the relation files and form tuples of the resulting relation which will be inserted into the relation file and the attribute values inserted into the access structure. The TID pair is completely processed before the processing of the first or second relation continues.

The processing of the second relation would continue until the key value becomes equal to the last value processed for the first relation. When this condition occurs the next key is processed for the first relation and the processing of the second relation starts from the first leaf again. This processing continues until the first relation processing has processed all the keys. This method of using the first relation as the control and repeatedly processing the second relation does not provide any duplicate tuples in the resulting relation.

<u>JOIN<</u>. The JOIN< operator selects the tuples from the source relations where the attribute value of the first relation is less than the attribute value of the second relation. The processing of the JOIN< operator is

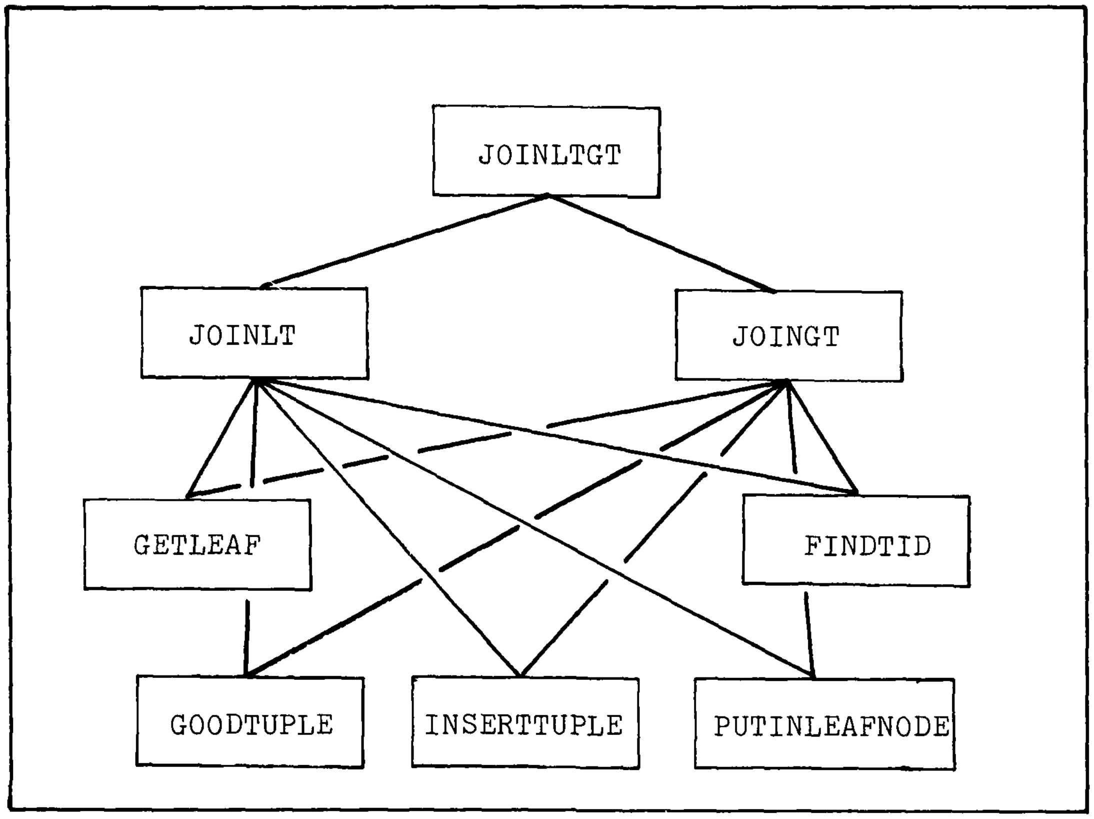


Figure 26. Basic Structure of JOINLTGT

FINDTID searches the leaf page for a TID header record that contains the relation ID of the first source relation and the attribute ID of the chosen attribute. If an appropriate TID header record is not in the leaf page, then the next key record is read and its leaf page searched. When a TID header record is found for the first relation, the TIDs associated with the TID header are stored and the value of the key is kept.

The processing of the second relation now begins by reading the first leaf and reading the first key record. The value of the key is compared to the value retained from used to access the source relations and produce the tuple of the resulting relation which would be inserted as described previously.

The access structure chosen provides an easy method of providing the equi-join. The TIDs for the same attribute value are stored in the same leaf page. So the leaf pages only need to be searched for the first relation identifier in a leaf page. If the first relation identifier is found, then the leaf page is searched for the second relation identifier. If the second relation is found then the TIDs for both relations are retrieved and the tuples of each relation retrieved and combined to form the tuples of the new relation. Again, each tuple from the first relation is combined with each tuple of the second relation only once to insure no duplicate tuples in the resulting relation.

The JOIN> and JOIN< operators are very similar. Therefore, they were combined in one module called JOINLTGT to share common procedures. Figure 26 shows the structure of the JOINLTGT module. The following paragraphs discuss the actual operators JOIN> and JOIN<.

<u>JOIN></u>. The JOIN> operator selects the tuples from source relations where the chosen attribute value of the first relation has a value greater than the second relation's chosen attribute value. The algorithm accomplishes this by getting the first leaf. It reads the first key and uses the pointer to the leaf page to call FINDTID.

FINDTID is then called to determine if a TID header exists for the first source relation and its chosen attribute. If there is a TID header record present for the first relation, the rest of the leaf page is searched to see if there is a TID header for the second source relation. If the second relation does have a TID header, the condition is satisfied so each TID that is associated with the first TID header is combined with each TID of the second TID header to form a TID pair.

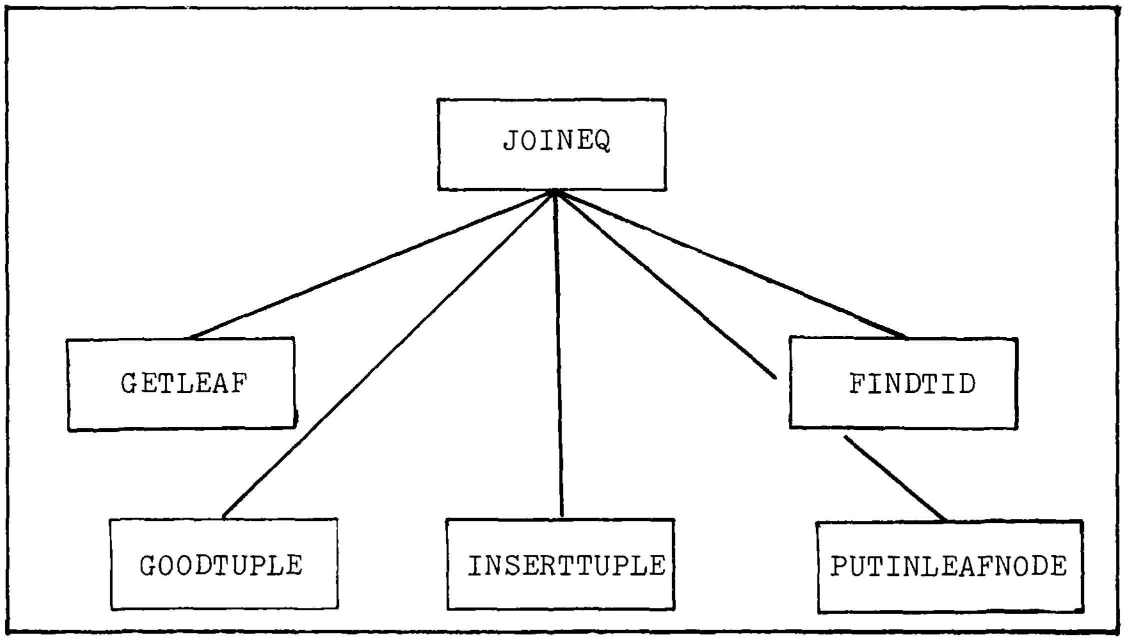


Figure 25. Basic Structure of JOINEQ

This TID pair gives the tuple identifiers of a tuple from each relation that when put together will form a tuple in the resulting relation. If either of the TID headers were not found, then the search in that leaf page is stopped and the next key record is read. The TID pair can then be means that each tuple of the control function is compared to all the tuples of the other input relation to see if the join condition is satisfied.

After all the tuples of the other input relation are compared, the next tuple of the control function is retrieved and the process begins again. This means that each tuple in the control relation is compared to the tuples of the other input relation only once. Since all of the tuples in both input relations were unique, combining tuples from the input relations will not cause duplicate tuples unless the same pair of tuples in combined twice. But the use of the control procedure only allows a tuple from the control relation to be used once with each tuple of the other relation; thereby, eliminating any possible duplicate tuples in the resulting relation.

JOIN=. The JOIN= operator selects the tuples from the source relation where the chosen attributes have the same value. The module JOINEQ is called to provide the JOIN= operator. Figure 25 shows the structure of the JOINEQ module. To find the same value of the attributes, the first leaf for the defined domain is read into the buffer (remember the first leaf is block 0). The first key is then accessed to provide the leaf page pointer. The relation IDs of the source relation are compared and the smaller ID is used first since the TID headers are stored in an ascending order based on the relation ID and then the attribute ID.

conjunction with the selection criteria to provide a list of TIDs. The last task is to delete the tuple associated with each TID from the relation file and delete the attribute values from the leaf files and if necessary call the B-tree delete module.

The procedures that validate the relation name and validate the user as a valid user for the given function had been tested in the INSERT module test. So the building of the selection criteria from the user's input was the first function to be tested. This test is a self documenting test since the module displays the selection criteria for validation by the user. Next, the selection criteria had to be used to select the appropriate TIDs from the access structure.

The testing of the remaining tasks of the DELETE module was done in two phases. The first phase just tested the capability to select the appropriate TIDs with the selection criteria. The second phase of testing then took the TIDs and performed the necessary deletion from the relation file, leaf files, and B-trees.

The GENSELECTLIST is the module that contains the algorithms to implement the selection criteria. To provide a complete test of the module, several different test cases were run using the FREQUENT relation. The first case was find all TIDs where DRINKER = TOM. This case tested the equal selection phase. Next, a single less-than condition and a single greater-than condition were tested. When each of the single conditions was successfully tested, the case of a compound condition using an OR was tested and this was followed by the test of the AND condition. The final test was to provide a compound criteria for one attribute and provide a compound criteria for a second attribute. This causes the selection criteria to combine selected lists from separate attributes, thus testing the final process of the GENSELECTLIST module. The results of these tests were manually matched against the tuples in FREQUENTS to insure proper selection. This completed the testing.

Now the second phase of the testing could begin. The first step was to use the first TID and retrieve the tuple associated with it. This tuple is then displayed to the user to allow the user to validate that this tuple should be deleted. This also insured that the tuple was retrieved correctly from the relation file.

The selection criteria were carefully selected so the tuples selected tested the following conditions when the attribute values were deleted from the leaf file.

1. Delete the largest value of a leaf which causes the value for the leaf to be changed in the B-tree.

2. Delete a leaf value that is not the largest value in the leaf to insure the key records are properly moved to eliminate the deleted key record. This also tested the ability to delete not only a TID record from the leaf page but also the TID header.

3. Delete a TID value from an overflow leaf page.

4. Delete a complete leaf. This required the delete algorithm to properly maintain the next and previous

leaf pointers in the leaf that sequentially fall before and after the deleted leaf. Also, this caused the B-tree delete function to be tested in the removal of a B-tree record from a B-tree node.

These test cases were completed to verify that the DELETE module did perform correctly. Again, the B-tree delete procedures were not thoroughly tested. Further work with the system should thoroughly verify that the B-tree delete functions do work correctly for all cases.

MODIFY Module. The verification of the MODIFY module was almost complete when the INSERT and DELETE modules were verified. The only procedure that had not been previously tested was modifying the tuple in the relation file. Therefore, the test cases were primarily tests to insure that the tuple was correctly modified in the relation file and then the appropriate attribute values were deleted from the leaf file and the new attribute values from the modified tuple were inserted.

Retrieve Module Testing

The Retrieve module consists of the many modules that implement the ability to retrieve data from the database. Among the functions of the Retrieve module are the functions to edit and manipulate a file of queries. Also, the ability to display the tuples of a relation is included as a function of the Retrieve module. These functions did not require intensive testing since their results are not critical to other procedures and the results are all self documenting.

The critical modules of the Retrieve module are the Optimize module and the Run module. The Run module consists of the procedures to implement the relational operators and perform the queries. The Optimize module performs the query optimization as described by Roth (6). Previous verification of the optimize module had been done, so the only testing done was to validate that the optimization routines did work as prescribed. This validation was attempted with the following queries:

```
JOIN FREQUENTS, DRINKER_DATA WHERE DRINKER = DRINKER
      GIVING rel1
PROJECT rel1 OVER FREQUENTS-DRINKER,ADDRESS,BAR
     GIVING rel2
SELECT ALL FROM rel2 WHERE
     FREQUENTS-DRINKER > PETE GIVING rel3
```

When the Optimization module started the procedure of pushing the select down the query tree, the system responded with a "recursive stack overflow" message. Several attempts were made to try to redefine the procedures called into different overlays to overcome this problem, but no solution was found. Therefore, the Optimization routine procedures were not fully validated. Since the primary focus during this thesis effort was to make the system operational, the individual procedures that implement the relational operators were tested to verify that the system was operational in a limited version. Also, when the recursion stack problem can be fixed, the system should become fully operational since the procedures to implement the relational operators will have been verified.

The primary focus of the testing of the relational operators was that the operator could correctly define the resulting relation and then select the correct information to be inserted into the resulting relation. The insertion procedures for all of the relational operators are the same procedures used to implement the INSERT module so they had been previously verified. Therefore, the insert procedures were not scrutinized as closely as the other procedures mentioned above.

The following are the relations and their tuples used during the testing of the relational operators.

Relation = FREQUENT

	DRINKER	BAR	BEER
#1	AL	45	45.366
#2	TIM	45	45.666
#3	TOM	44	35.466
#4	TOM	45	45.366
#5	ZURD	45	45.98

Relation = DRINKER_DATA

	DRINKER	ADDRESS	COMMENT
#1	ALICE	WHO KNOWS	NONE
#2	JOE CO[illegible]	454545 LIBRARY DESK	STUDYSTUDY
#3	TIM	1122334455 AFIT ROAD	NO COMMENT
#4	TOM	4545 NATIONAL LANE	Tom's Comment
#5	TOM K	6767 WHO CARES LANE	NO COMMENT
#6	ZURD	23 AFIT 23	NONE

The relations used were the relations defined during the test of the data definition module. The information contained in the relations is nonsensical data made up only for test purposes. The data does provide some good test cases because some very similar attribute values are used. This was done to insure that the selection modules can distinguish between names such as TOM and TOM K. The first relational operator tested was SELECT.

SELECT. The SELECT module was thoroughly tested by using five different test cases. The first case tested the ability to select when the condition given was an equal condition. The condition used was

```
SELECT ALL FROM DRINKER_DATA WHERE DRINKER = TOM
     GIVING rel1
```

This query required the module to define the new relation, rel1, as a temporary relation in the data dictionary before actually processing the query. The definition phase was carefully examined by using the INVENTORY module to insure the definition was correct. The result from this query was the single tuple, TOM, 4545 NATIONAL LANE, Tom's Comment. This verified that the selection process was working correctly since the tuple with DRINKER = TOM K was not selected even though it had the desired value as part of its value.

The test cases that followed checked the > condition, the < condition, combining two selects with an AND, and combining two selects with an OR. The cases using the AND and OR were tested with the selection criteria selecting values both from the same attribute and from different attributes. The following are some of the test cases successfully run:

```
1. SELECT ALL FROM FREQUENTS WHERE
        DRINKER > TIM AND DRINKER < ZURD GIVING REL1
2. SELECT ALL FROM DRINKER_DATA WHERE
        ADDRESS > 45 OR NAME < KURT GIVING REL1
3. SELECT ALL FROM FREQUENTS WHERE
        BEER = 45.366 AND BAR < 45 GIVING REL1
```

The result of the first select was the tuples: TOM, 44, 35.466 and TOM, 45, 45.366. The result of the second select with multiple conditions was the following tuples: ALICE, WHO KNOWS, NONE; JOE COOL, 454545 LIBRARY DESK, STUDYSTUDY; TOM, 4545 NATIONAL LANE, Tom's Comment; and TOM K, 6767 WHO CARES LANE, NO COMMENT. The duplicate tuple was removed thus showing that the OR combination was working successfully. The last multiple condition test returned a relation with no tuples which showed that the AND condition was successfully combining the results from the separate conditions to form the compound condition.

<u>PROJECT Module</u>. The PROJECT module was tested with the following cases:

1. PROJECT FREQUENTS OVER DRINKER GIVING REL1
2. PROJECT FREQUENTS OVER BAR GIVING REL1
3. PROJECT FREQUENTS OVER DRINKER,BAR GIVING REL1
4. PROJECT FREQUENTS OVER BEER, BAR GIVING REL1.

The first two tests were to test the ability of the routine to recognize and eliminate the duplicate tuples. The third test was to insure that more than one element could be handled correctly. The final test was to insure that the procedure would insert the attribute values in the correct order in the tuple since this is a different ordering of the attributes than is contained in the source relation.

The first result checked was the data dictionary definition of the resulting relation. When the definition was verified correct, then the remaining portion of the PROJECT module was allowed to execute. The results of the final case are shown below to illustrate the results of the PROJECT module.

```
          Relation = REL1
          BEER              BAR
---------------------------------------------
     #1   35.466            44
     #2   45.366            45
     #3   45.666            45
     #4   45.98             45
```

<u>JOIN</u> <u>Module</u>. The JOIN module actually has three different joins. The three joins are the JOIN=, JOIN< and the JOIN>. The first test of the JOIN module was to insure that the resulting relation was correctly defined as a temporary relation in the data dictionary and the name of each attribute was unique. This test was performed with the query:

```
     JOIN FREQUENTS, DRINKER_DATA
          WHERE DRINKER = DRINKER GIVING REL1.
```

A join with an equal condition was used here but that did not matter for this test since only the data dictionary definition was being checked.

The results in the data dictionary did show that the REL1 was correctly defined and that the two attributes with the name DRINKER were correctly identified as FREQUENTS-DRINKER and DRINKER_DATA-DRINKER. Another test was done where a relation was joined with itself to insure that the unique name generator did work correctly. This time all of the names were the same even after adding the relation name. Therefore, the names had to be suffixed with a -1 and -2 to make them unique. The next step in the testing of the JOIN module was to test the individual join operators.

JOIN=. The equal condition of the join was tested with the following query.

```
    JOIN FREQUENTS, DRINKER_DATA WHERE
         DRINKER = DRINKER GIVING REL1
```

The procedure has to first select the tuples from each rela.ion that satisfy the equal condition and then combine the tuples from the two relations into a new tuple for the resulting relation. The results from this query were:

```
         Relation = REL1
    FREQUENTS-DRINKER   BAR        BEER
         DRINKER_DATA-DRINKER ADDRESS       COMMENT
-------------------------------------------------------------
#1  TIM                    45         45.666
         TIM           1122334455 AFIT ROAD  NO COMMENT
#2  TOM                    44         35.466
         TOM           4545 NATIONAL LANE    Tom's Comment
#3  TOM                    45         45.366
         TOM           4545 NATIONAL LANE    Tom's Comment
#4  ZURD                   45         45.98
         ZURD          23 AFIT 23            NONE
```

The results of this test show how the tuples were combined to form a tuple of the new relation. The results also showed that the procedure correctly combined both tuples of the first relation that contained TOM with the single tuple in the second relation. This tested the equal condition, therefore the basic principles of the join operator had been verified but now the other conditions needed to be tested.

JOIN>. The JOIN> condition was tested with the following query.

```
     JOIN DRINKER_DATA, FREQUENTS WHERE
          DRINKER > DRINKER GIVING REL1
```

This a join with a condition where the value of the attribute in the first relation should be greater than the attribute value in the second relation for the tuples to be selected. The results of this test were:

```
          Relation = REL1
     DRINKER_DATA-DRINKER  ADDRESS       COMMENT
               FREQUENTS-DRINKER   BAR       BEER
--------------------------------------------------------------
#1   ALICE                 WHO KNOWS        NONE
               AL                        45      45.366
#2   JOE COOL              454545 LIBRARY   STUDYSTUDY
               AL                        45      45.366
#3   TIM                   1122334455 AFIT ROAD  NO COMMENT
               AL                        45      45.366
#4   TOM                   4545 NATIONAL LANE    Tom's Comment
               AL                        45      45.366
#5   TOM                   4545 NATIONAL LANE    Tom's Comment
               TIM                       45      45.666
#6   TOM K                 6767 WHO CARES LANE   NO COMMENT
               AL                        45      45.366
#7   TOM K                 6767 WHO CARES LANE   NO COMMENT
               TIM                       45      45.666
#8   TOM K                 6767 WHO CARES LANE   NO COMMENT
               TOM                       44      35.466
#9   TOM K                 6767 WHO CARES LANE   NO COMMENT
               TOM                       45      45.366
#10  ZURD                  23 AFIT 23            NONE
               AL                        45      45.366
#11  ZURD                  23 AFIT 23            NONE
               TIM                       45      45.666
#12  ZURD                  23 AFIT 23            NONE
               TOM                       44      35.466
#13  ZURD                  23 AFIT 23            NONE
               TOM                       45      45.366
```

The results showed the selection criteria and combination algorithm both worked to create the tuples of the new relation for the JOIN>.

<u>JOIN<</u>. The JOIN< condition is the condition when the attribute value of the first relation has to be less than the attribute value of the second relation for the

tuples to satisfy the selection condition and be combined to form a tuple in the resulting relation. The query used to test this condition was the following query.

```
    JOIN FREQUENTS, DRINKER_DATA WHERE
         DRINKER < DRINKER GIVING REL1
```

The results of this test were the same as the test for the JOIN> except the FREQUENT part of each tuple was listed first in the resulting tuple. This showed that the JOIN< and all the JOIN modules do perform correctly.

Summary

The testing of the system was performed in a very limited environment because the memory space for the data dictionary and other linked lists was very limited. That is why only two relations were defined and very limited amounts of data were in each relation. Because of the memory limitations, the complete validation of the system could not be completed, although most operators and functions were validated.

The processing time of some processes was excessive because of the great amount of overlaying of memory necessary. The overlaying is necessary to provide at least a small amount of memory for the data dictionary and other dynamic memory allocations used in the system. Even with all of the limitations of time and memory space, the testing of the system proved that the AFIT relational database system was operational for limited applications.

V. Conclusions

The goal of this thesis was to design and implement a relational database on a microcomputer. This specifically involved the design of a low level access structure and implementing the relational operators. The low level access structure was implemented but during the testing of the edit modules it became obvious that processing time was becoming a problem.

The factor of slow processing time was the result of having insufficient memory in the microcomputer (64K). The insufficient memory caused the use of numerous program overlays to be able to make the system functional. The overlays created a great deal of system overhead to handle the reading of the overlays from disk and placing the overlays in memory. Thus, the result was very slow processing time.

The shortage of memory also caused the relational operators that were implemented to be tested only in a very limited environment of very small amounts of data. Also, the optimization of the queries could not be tested because of the lack of memory space. Thus, no validation of the optimization routines could be performed. Therefore, no analysis could be performed on the efficiency of the optimizied queries performance versus the unoptimized queries performance. Thus, there are several areas that remain for future research.

Recommendations

Due to the limited computer and time environment in which this development was conducted, there are many areas that need further development or research. These recommendations are:

(1) Implement the DIVIDE, UNION, DIFFERENCE, PRODUCT, and INTERSECT relational operators to provide the complete set of relational operators described by Roth (6).

(2) Determine the amount of data necessary in a relation to make it more efficient to use the access structure rather than just accessing the relation directly.

(3) Develop reorganization algorithms to eliminate the wasted space in the relation and leaf files created when records are deleted or redefine the leaf and relation file structures.

(4) Convert the data dictionary from the linked list structure to relations stored in the database and implement any special access routines for initializing the system.

(5) Develop some scheme for backup capabilities of the relation, B-tree, and leaf files to allow recovery from a system crash during a data modification operation.

(6) Consider the possibility of developing a different set of relational operators that operate on the intermediate relations of a query tree to provide faster processing time of queries. This should include a study of the possible use of the co-ordinating operator constructor's preferred sort orders in processing the intermediate relations.

(7) Provide the ability to transfer data from one relation to another allowing the data from a temporary relation to be saved for later use.

(8) Investigate possible alternatives to the access structure now used to see if a more efficient access method might be possible.

(9) Insure the data integrity of the relation during the modification of data in the MODIFY operation.

(10) Explore the concept of including the insert, delete, and modify operations in the relational operators.

(11) Develop batch interface capabilities so that another computer could interface more directly to the system.

(12) Overcome the memory overlay problem by converting to a computer with more memory. Some possible solutions are:

a) Convert to LSI-11/23 with extended memory. This would be compatible with other work done.

b) Move to Z-100 with 16-bit Pascal under MS-DOS. This would make it compatible with the government standard microcomputer.

c) Move to VAX to complete the implementation of the system. This would allow complete analysis of the performance of the system but does lose the objective of being microcomputer-based.

(13) Perform extensive performance analysis of the system to include instrumenting the code, providing a

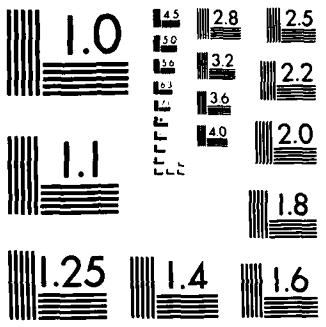
AD-A151 770 IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G KEARNS 2/2

UNCLASSIFIED DEC 84 AFIT/GCS/ENG/84D-12 F/G 9/2 NL

END

FILMED

DTIC


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

benchmark database, and doing simulation and modelling of the system.

(14) Extend the system to include

- subschemas
- multi-user capability
- network DBMS mode
- connection as backend to a host

Conclusion

The implementation of the AFIT relational database on a microcomputer was successful. However, the implementation has limitations. Some of the limitations are: poor performance due to the requirement for numerous program overlays, insufficient memory space for query optimization, and lack of complete implementation of all relational operators. In spite of the limitations, the AFIT relational database system does provide an operational relational database management system; however, the system's real value is as a pedagogical database for future students.

Appendix A: Pascal/MT+

Pascal/MT+, version 5.5, was the system used to support the development of the relational database system. Pascal/MT+ supports all of the standard Pascal features and has some extensions to the standard Pascal. The extensions to the standard Pascal used during this development and some of the problems encountered with Pascal/MT+ will be discussed.

The Pascal/MT+ system operates on a machine using the CP/M operating system, version 2.0 or later. The Pascal/MT+ system was selected for development because it very closely matched UCSD Pascal which had been used for previous development. But Pascal/MT+ provides greater ability to use program overlays, thus somewhat relieving the memory space problem encountered during previous development of the relational database system.

Pascal/MT+ Unique Features

The features described below are not all of the unique features supported by the Pascal/MT+ compiler; only the features used during this development. The main feature of Pascal/MT+ used was its string handling capability. A string in Pascal/MT+ is a defined type. The string is like a packed array of characters in which byte 0 contains the dynamic length of the string and bytes 1 through n contain characters. The default length of the string is 80 but may be defined to be from 1 to 255 characters in length.

The string type also allows the comparison of two strings even though the strings are defined as different lengths. The functions described below are provided by Pascal/MT+ for handling strings. It should be noted that these functions are almost identical to the string handling functions found in UCSD Pascal.

First, the syntax of the procedure or function will be shown and then a brief description of its action will be provided.

FUNCTION LENGTH(STRING):INTEGER;
This function will return the intege value of the length of the string.

FUNCTION CONCAT(SOURCE-A,SOURCE-B,...,SOURCE-N):STRING;
This function returns a string in which all of the sources are concatenated. The sources may be string variables, string literals, or characters.

FUNCTION POS(PATTERN,STRING):INTEGER;
This function returns the integer value of the position of the first occurrence of PATTERN in the STRING. STRING is a string and PATTERN can be a string, a character, or a literal.

PROCEDURE DELETE(STRING,INDEX,SIZE);
This function removes SIZE characters from STRING starting at the byte named by INDEX. STRING is a string. SIZE and INDEX are integers.

Pascal/MT+ supports a form of random file access that is supported by CP/M, version 2.0 and later. This ability to randomly access files has been used in the development of this database system. Also, the ability to extend files or append to files is supported by Pascal/MT+. This capability relieves the need for most of the overflow file fields built in the data structures. The overflow fields were left in the data structures to allow for the possibility of later transfer to a system that does not allow dynamic file growth and overflow files would be needed. The following procedures were used for file handling.

PROCEDURE ASSIGN(FILE,NAME);
    This procedure assigns an external file name to a file. FILE is a defined file name of any file type. NAME is a literal or variable string containing the name of the file.

PROCEDURE BLOCKREAD(FILE,BUFFER,IOR,SIZE,RB);
PROCEDURE BLOCKWRITE(FILE,BUFFER,IOR,SIZE,RB);
    These procedures provide direct CP/M disk access. FILE is an untyped file. BUFFER is an array of characters which is large enough to hold the data. IOR is an integer which receives the returned value from the operating system indicating if the operation was successful or indicating the error that occurred. SIZE is the number of bytes to be transferred and must be a multiple of 128. SIZE and BUFFER are related in size because BUFFER must be as large as SIZE. RB is the relative block number of the file. Each block is 128 characters. Since 512 characters is 4 blocks, if 512 characters are written to a file at relative block 0 the next time data is added to the file it would need to start at block 4 so as not to destroy the previous data. The data is transferred to or from the users BUFFER variable for the specified number of bytes.

PROCEDURE CLOSE(FILE,RESULT);
    This procedure closes FILE and returns the integer RESULT. FILE is a file. This procedure guarantees that data written to a file is properly purged from the file buffer to the disk. RESULT is an integer returned by the operating system to indicate if the CLOSE was successfully completed or if an error occurred in closing the file.

FUNCTION IORESULT : INTEGER;
    The IORESULT returns an integer from the operating system which is altered after every CP/M file access. This integer indicates if the file operation was successful or provides an indication as to what the error was.

Compiling and Linking Programs

Pascal/MT+ allows modular compilation of separate modules and also provides the user the ability to use program overlays. To be able to use these capabilities, the programs written using Pascal/MT+ contain some extra words and numbers not normally seen in a Pascal program. One of the reasons Pascal/MT+ was used was for its ability to provide memory overlays in execution. This allows the computer to store the unused program segments on disk rather than in memory. The first thing used to provide overlays is a modular approach. This means that one main program is written and then modules are added. The main program will reside in memory at all times while the program executes and the modules are overlayed in main memory.

The program requires one extra type of statement to allow this modular approach. This is a statement called EXTERNAL. Any procedures or functions that are referenced in the main program must either be contained in the main program or have an EXTERNAL statement that provides the header for the procedure or function. Included in this EXTERNAL statement is a number in brackets. This number provides the overlay number in which the procedure can be found. An example of the EXTERNAL statement would be:

```
EXTERNAL [15] PROCEDURE TEST(X:INTEGER;DONE:BOOLEAN);
```

The modules that are not included in the main programs have to have the statement MODULE and a name as the first statement in order to compile. The module's last statement is MODEND. These statements are equivalent to the PROGRAM and END in the main program segment. The module may reference procedures and functions not included in the module. But just as in the main program, any functions or procedures not included in the module must have an EXTERNAL statement to name them. The modules may call functions or procedures that are in other modules. This also means that one overlay may call another overlay.

The Pascal/MT+ system provides the linker to link these compiled modules into a single program or into overlays. Since extensive use of overlays was necessary, a brief description of the commands used to link the modules into overlays will be provided. The following example shows the commands necessary to link a program and three overlays:

```
1. LINKMT MAIN,FPREALS/S,PASLIB/S/D:8000/X:2000/V1:4000/V2:6500
2. LINKMT MAIN=MAIN/O:2,OVERLAY1,PASLIB/S/P:4000/X:100
3. LINKMT MAIN=MAIN/O:B,OVERLAY2,PASLIB/S/P:4000/X:900
4. LINKMT MAIN=MAIN/O:12,OVERLAY3,OVERLAY4,PASLIB/S/P:6500/X:50
```

The first command links the main program. The extra files FPREALS and PASLIB are Pascal/MT+ libraries that contain run-time functions. The /D:8000 indicates that the program data area should start at 8000 hexidecimal. Note that all the numbers expressed in the link commands are in hexidecimal. /X:2000 means that the heap should start 2000 hexidecimal bytes from the end of the main program data area. This space is used to provide the data areas for the overlays. The next two parameters in the first command are the overlay area indicators. The V1 indicates that all overlays in overlay group 1 will be loaded into memory starting at memory location 4000 hex. Overlay group 1 includes overlays 0 thru 15 decimal as they are numbered in the EXTERNAL statements or 0 thru F as they are numbered in the link commands. V2 indicates the starting position for overlay group 2. Any overlay group not represented by a /V parameter receives the same starting position as overlay group 1.

Commands 2 thru 4 show the command used to link the overlay modules. The MAIN=MAIN/O:* parameter names the name of the main program segment. The /O:* states that this is an overlay and the hexidecimal number that goes in the place of the asterisk indicates the number of the overlay. Remember that the overlay number is in *hexidecimal here* but in the EXTERNAL statements in the code the numbers are in decimal. Next, the name of the module or modules to be used to create this overlay are listed. Following that the run-time library is searched for any run-time procedures that might need to be included. The /S following the name of the library means that only the modules of the library that are needed are used. /P indicates the starting position for the module. This has to match the appropriate /V parameter in the first link command. The /X parameter is supposed to allow an offset from the end of the main program's data area for the overlay's data area.

The /X command was found to be nonfunctional in the overlay link commands. Thus, if one overlay called another overlay their data was stored in the same location. This problem plagued the development effort for several weeks before it was determined that this was the cause of the problem. The method used to counteract this problem was to include a dummy procedure with a dummy array of characters at the start of each overlay module. By including and adjusting the size of the array, this problem was overcome. An example of the procedure that was added to each module follows:

```
     PROCEDURE DUM;
     VAR DUMMY:ARRAY[0..1000] OF CHAR;
     BEGIN END;
```

The array size has to be adjusted by using the data size returned when a program is linked but remember the number returned from the linker is in hexidecimal and should be converted to decimal for this dummy array to be correctly dimensioned. If a new version of the linker is obtained that has this problem fixed, these dummy procedures should be removed.

The one remaining feature that should be mentioned is the compiler toggles. Several compiler toggles are used in the code to provide a flag to the compiler during compilation. These flags all should be on a line by themselves and they start with (*$ and end with *). The important ones used are (*$S+*) which when used as the first line in a program or module (before the word PROGRAM or MODULE) indicates that this is a recursive routine or contains recursive routines and (*$E+*) or (*E-*). (*$E-*) tells the compiler to not put the names of the following procedures in the external name table. This means that the names following the (*$E-*) cannot be used in an EXTERNAL statement. (*$E+*) restores the compiler to its default condition of placing all procedures names in the external name table.

The features described above are some of the unique features of Pascal/MT+. These features make Pascal/MT+ very similar to UCSD Pascal, the language from which the system was being converted. But UCSD had some features that did not have a command in Pascal/MT+ that would exactly map to the UCSD command. The two important commands from UCSD that did not have a corresponding command in Pascal/MT+ were MARK and RELEASE.

MARK and RELEASE are UCSD Pascal's method of being able to return memory that has been dynamically allocated during execution. MARK is used first before the memory is allocated. It stores a memory address that is the current memory address from where the dynamic memory or heap will grow. RELEASE takes the memory address and returns to the system all of the heap above this address. This does not allow for only returning individual segments in the heap. To allow these commands to work in Pascal/MT+, new functions were constructed that performed these same functions by retrieving the top of heap address from the system and then storing this value using the MARK. RELEASE replaced the current top of heap value stored in the system by the value stored during the MARK command.

The MARK and RELEASE command were implemented to ease the conversion from UCSD Pascal to Pascal/MT+. However, Pascal/MT+ does provide the standard heap management function DISPOSE for returning individual segments of the heap. Also in the attempt to keep the code as close to its original form, a function called GOTOXY was implemented. This function is a function provided in UCSD Pascal but not provided in Pascal/MT+. It is included in a group of screen manipulation functions that are CRT dependent and have to be modified for each different type of CRT used. These functions are included in the module COMON4.

The functions, procedures, and features described above, describe some of the unique elements of Pascal/MT+ that were encountered during the development of the relational database system. Also, some functions that were implem   ed to make Pascal/MT+ more closely resemble UCSD Pascal were briefly discussed. These features were discussed to attempt to make the code used more understandable to a person who has never used Pascal/MT+.

Appendix B: B-tree Access Structure Concept

The B-tree is a generalization of a binary search tree that was introduced in the late 1960s. The B-tree of order m is a m-way balanced search tree. To be a B-tree it must also meet the following conditions:

1) The root is either a leaf or has at least two children.
2) All nodes other than the root and the leaves must be at least half full.
3) Each path from the root to a leaf must have the same length.

There are many forms of B-trees but one common name for a variation of the B-tree is B*-tree.

The specific form of B-tree used in the AFIT relational database is called a B+-tree. There is much confusion about the names of various forms of B-trees so sometimes it is referred to as a B*-tree but a B*-tree requires that each node be at least 2/3 full, not just 1/2 full. The concept of the B+-tree is that all the keys reside in the leaves. This means that the upper levels of the B+-tree consist only of an index to the leaves.

The advantage of the B+-tree is that the upper levels enable rapid location of the index and key parts. But it also allows the leaves of the tree to be linked through the use of next or previous pointers to allow a sequence set.

This means that for random access to an individual key the index can be used to provide the key quickly but if the complete set of keys needs to be processed sequentially, the sequence set can be used to access the keys sequentially. These two features combined provide a very powerful indexing method. Figure B-1 shows an example of a B+-tree.

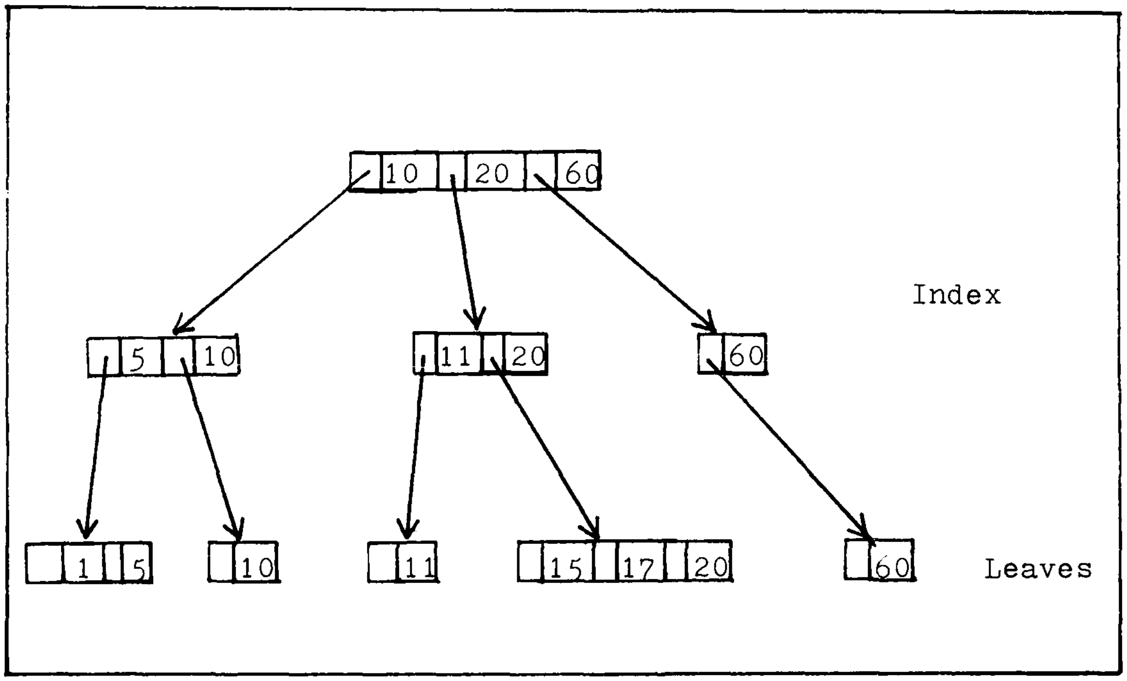


Figure B-1. Example of a B+-tree

The concept of the B+-tree used in the AFIT relational database also included the idea that each key in a leaf point to another record called a leaf page that contains all the information about the key. In the relational database, the information about the key is all of the TIDs of the crocomputer. This was accomplished by implementing the access structure concept, the relation edit functions, and the limited set of relational operators using Pascal/MT+ on a 64K CP/M computer. Although the system is operational, it is currently limited in its effectiveness by a lack of memory space in the computer. In spite of the limitations, the AFIT relational database does provide a valuable pedagogical relational database management system for future student research.

References

1. Kearns, Capt Timothy G. Implementation and Analysis of a Microcomputer Based Relational Database System, MS Thesis AFIT/GCS/ENG/84D-12. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984.

2. Rodgers, 2LT Linda M. The Continued Design and Implementation of a Relational Database System, MS Thesis GCS/EE/82-29. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982 (AD-A124 927).

3. Roth, 2LT Mark A. The Design and Implementation of a Pedagogical Relational Database System, MS Thesis GCS/EE/79-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1979 (AD-A080 240).

4. Smith, John Miles and Philip Yen-Tang Chang, "Optimizing the Performance of a Relational Algebra Database Interface," Communications of the ACM, 18: 568-579 (October 1975).

The JOIN operator is a combination operator. This means that it selects tuples from two input relations and combines the tuples to form a tuple of the result relation. The selection criteria of the JOIN operator allows the two input relations to be joined where the designated attributes are equal, less than, or greater than. This means that the attributes from each input relation must be defined on the same domain. Also, the JOIN operator does limit the criteria to be a single condition.

The JOIN operator uses the first attribute in the selection criteria as a control. This means that for each attribute value of the first relation all attribute values of the second input relation are compared. If the values compared satisfy the selection criteria, the tuple from the first relation and the tuple from the second relation are combined to form a tuple of the result relation. The tuple is then inserted into the result relation file and its attribute values inserted into the proper access structures. This method of using one value as a control eliminates introducing duplicate tuples in the result relation; thus, tuples inserted into the result relation during the JOIN operation do not have to be checked to see if they are duplicates.

Summary

The purpose of the research described was to provide an operational relational database management system on a mi-

Implementation of the Relational Operators

The relational operators that were implemented are PROJECT, SELECT, and JOIN. These are the basic operators of relational. Other operators in the relational algebra are PRODUCT, DIFFERENCE, UNION, DIVIDE, and INTERSECTION. These operators are to be implemented at a later date.

The PROJECT operator takes only selected attributes of a relation to make a new relation. To implement this function each tuple is read from the relation file. The selected attributes of the tuple are read, and the values of the selected attributes are combined into a new tuple. The new tuple is then inserted into the result relation file and each attribute value inserted into the appropriate access structure. The access structure is not used to find the TIDs of the input relation because the order in which the relation is processed is not critical as long as each tuple of the relation is processed.

The SELECT operator selects only specified tuples from a relation and uses the selected tuples to create the result relation. The implementation of the SELECT operator uses the criteria evaluation procedures of the delete function of the relation edit functions to produce a list of TIDs that identify tuples that satisfy the selection criteria. Then each tuple is accessed using its TID. The tuple is then inserted into the result relation and each of the attribute values is inserted into the access structure with the TID from the result relation.

Then each attribute value with the tuple's TID is inserted into the access structure of its domain.

The deletion function receives a list of criteria used to select tuples to be deleted. This list evaluates each criterion individually by searching the appropriate access search for TIDs that satisfy the condition. The TIDs that satisfy the condition are then placed in a linked list. As each criterion is evaluated, the previous linked list of TIDs is either combined with the new list by "anding" or "oring" the lists. After all the criteria have been evaluated the remaining linked list of TIDs identifies the tuples of the relation to be deleted.

Each tuple identified is then read from the relation file and flagged as deleted in the relation file. Also, each attribute value of the tuple is used to delete the attribute value and TID combination from the access structure.

The final edit function is the modify function. It combines both the operations of the delete and insert functions to find the tuples to be changed, reads the tuple, changes the necessary values of the tuple, replaces the tuple in the relation file, deletes the attribute values that are going to be changed from the access structures, and inserts the changed attribute values into the appropriate access structures.

checking to see if the user is the owner of the relation involved and if the user is not the owner, the system requires the user to provide a correct password before continuing. Once the user has passed the security test, the function has the user interactively build a list of tuples to be inserted, a set of criteria used to select tuples to be deleted, or a set of criteria used to select tuples to be modified with a list of the modification values.

The insert function considers each tuple to be inserted individually. It first retrieves the names of the attributes that form the key for the relation from the data dictionary. It takes the values of the key attributes, in the tuple to be inserted, and searches the access structure to see if that value exists in a tuple already in the relation. If the value does exist in a tuple or tuples of the relation, then the TIDs of the tuples are read from the access structure and formed into a linked list. This is repeated for each attribute in the key. The linked lists of TIDs for each attribute of the key are then compared. If any TID appears in every list the tuple to be inserted is a duplicate and will not be inserted.

After the tuple to be inserted is determined not to be a duplicate tuple, it is first inserted into the relation file. The insertion into the relation provides the TID for the tuple. The TID consists of the filename for the relation, the block number in the file where the tuple's data starts, and the offset in the block where the tuple starts.

Implementing the Leaf Structure

The design of the leaf structure considered the amount of disk space necessary, the number of disk accesses necessary, and the efficiency of inserting and deleting information from the structure. The resulting design was not the most efficient in any one area but time efficiency was the main consideration used.

The leaf structure provided for each leaf to contain key values and associated with each key value a pointer to a block in the file that contained all the tuple identifiers (TID) for that value. Leaf page was the name used for the blocks, that contain the TIDs, referenced from the leaves. Therefore each key in a leaf points to a leaf page where the TIDs for that value are stored. In order to provide sequential processing, the leaves were connected with previous and next pointers to provide a linked list of leaves. The key values in the leafs were also placed in a physical ascending order. The TIDs in the leaf pages are also maintained in an ascending order based on the ID of the relation and ID of the attribute within the relation. This ordering provides for more efficient searching of the leaf pages.

Implementation of the Relation Edit Functions

The relation edit functions include inserting tuples in a relation, deleting tuples from a relation, and modifying attribute values in existing tuples of a relation. The first step of each edit function insures the security by method suggested by Smith and Chang (4).

The basis for the relational database system was defined by Roth but the system still lacked the design and implementation of the necessary indexing or access structure to become an operational database management system. Linda M. Rodgers continued the development of the system by defining an access structure (3).

The access structure defined was based upon a B-tree structure to provide an index. It was defined that there would be a B-tree for each domain defined in the data dictionary of the system. Thus all attributes defined on the same domain would have their values indexed in the same B-tree. The B-tree would be a special type of B-tree where all the upper levels of the B-tree would just provide an index to the leaves where all the keys would be located. The leaves would then somehow indicate how to find the tuples that contained that attribute value. The index portion of the B-tree was implemented by Rodgers but the design of the leaves and how they should reference the tuples was undefined. The following paragraphs describe the design and implementation of the leaf structure, the implementation of the relation edit functions, and the implementation of the relational operators to provide a operational microcomputer based relational database system (1). All of the implementation described was accompilshed in Pascal/MT+ on a 64K CP/M system.

Appendix D: The Implementation of a Microcomputer Based Relational Database System

E. F. Codd first introduced the idea of utilizing the relational concept for data in 1970. Since that time, much has been written about the theoretical concepts of the relational view. Not only have the theoretical aspects of the relational view been studied but also the practical aspects have been widely researched. The advent of the microcomputer caused a demand for a relational database system that will run efficiently on this type machine. Therefore, in 1979, Mark A. Roth started the design and implementation of a relational database system for a microcomputer (2).

The intent of Roth's research was to provide a pedagogical database for student use at the Air Force Institute of Technology. The need for such a teaching tool to aid in the learning experience of database students had been identified by Dr. Thomas Hartrum of the AFIT/EN Electrical Engineering faculty.

Roth considered many key aspects of the relational database in his design and implementation. Some of these aspects were the means for data definition and data manipulation. The data manipulation research included selecting relational algebra as the basis for querying the database and designing a method of optimizing queries based upon a

```
A:LINKMT SPOS=SPOS/O:26,SPLIT,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:27,GETTUP,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:28,OPT4,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:29,OPTIMIZE,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2A,PROJ,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2B,SELECT,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2C,TRANSFER,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2D,JOINEQ,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2E,JOINLT,A:PASLIB/S/P:5000
```

The SPOS program has a library of run-time routines. This library is named GETLIB and was used in some of the previous link commands. The library is contained in the file GETLIB.ERL. The routines used to build the library are contained in the following files:

```
    GET1.PAS       GET2.PAS       GET3.PAS       GET4.PAS
    GET5.PAS       GET6.PAS       GET7.PAS       GET8.PAS
    GET9.PAS       GET10.PAS      GET11.PAS      GET12.PAS.
```

To create the library each of the .PAS files is compiled to produce .ERL files. Then use the following command to create the library.

```
    A:LIBMT GETLIB
```

This command uses a file named GETLIB.BLD as an input. GETLIB.BLD tells the library manager the name of the resulting library and what .ERL files are to be read to create the library. The library when created is linked with any modules that call for a procedure contained in the library. The advantage of using the library is the fact that only the necessary procedures are linked. This means that unused procedures are not linked which conserves memory space.

The procedures described above briefly describe the files and procedures necessary to modify the AFIT relational database system.

modify the code in the appropriate module and recompile that module. Then only that module needs to be relinked. The only time it is necessary to relink the complete set of modules is when a modification is made to SPOS.PAS or COMON4.PAS. The following list provides the commands necessary to relink the complete SPOS system. If an individual module is to be relinked just use the one command that contains the appropriate filename for the module.

Commands to link SPOS

```
-----------------------------------------------------------
A:LINKMT SPOS,COMON4,A:ROVLMGR,A:FPREALS/S,A:PASLIB/S/D:8000
/X:3100/V1:5000
  (Links main segement of program.  Following link overlays)
A:LINKMT SPOS=SPOS/O:1,SETUP,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:2,INVENT,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:3,USRET1,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:4,USEDIT,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:5,QUIT,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:6,PUTINB,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:7,DELETE,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:8,LEAFDL,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:9,DISPLAY,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:A,SELLIST,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:B,INDELMOD,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:C,DUM3,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:D,SIMSEL,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:E,OPT1,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:F,OPT2,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:10,OPT3,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:11,OPT5,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:12,OPT6,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:13,DUM4,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:14,RUN,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:16,TIDINS,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:17,IDEL,FINDLEAF,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:19,IEDIT,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:1A,IEDIT2,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:1C,EMPTY,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:1D,FULL,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:1E,FREE,GETLIB/S,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:21,INSTUP,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:22,TIDS,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:23,TIDFIND,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:24,LEAFS,A:PASLIB/S/P:5000
A:LINKMT SPOS=SPOS/O:25,LEAFFIND,A:PASLIB/S/P:5000
```

FPREALS and PASLIB are libraries of run-time procedures provided with the PASCAL/MT+ system. The linker will read the .ERL file of each file named and link them into a program named DBMGR.COM. Then to execute the program the user need only type DBMGR in response to the CP/M operating system prompt.

SPOS Modification

SPOS is the program that provides the data manipulation for the AFIT relational database system. The SPOS program is actually many modules that are linked in a special way to provide numerous overlays. The following is a list of the necessary files:

SPOS.PAS	COMMON.DEC	COMMON.PRC
DELETE.PAS	DISPLAY.PAS	DUM3.PAS
DUM4.PAS	EMPTY.PAS	FIXFIELD.PAS
FREE.PAS	FULL.PAS	FINDLEAF.PAS
GETS.PAS	GETTUP.PAS	IDEL.PAS
IEDIT.PAS	IEDIT2.PAS	INDELMOD.PAS
INSTUP.PAS	INVENT.PAS	JOINEQ.PAS
JOINLT.PAS	LEAFDL.PAS	LEAFFIND.PAS
LEAFS.PAS	MANNODE.PAS	OPT1.PAS
OPT2.PAS	OPT3.PAS	OPT4.PAS
OPT5.PAS	OPT6.PAS	OPTIMIZE.PAS
PRINTREE.PAS	PROJ.PAS	PUTINB.PAS
QUIT.PAS	RUN.PAS	SELECT.PAS
SELLIST.PAS	SETUP.PAS	SIMSEL.PAS
SPLIT.PAS	TIDFIND.PAS	TIDINS.PAS
TIDS.PAS	TRANSFER.PAS	USEDIT.PAS
USRET1.PAS		

The following files are not necessary for the system at this time but contain the code for the coordinating operator constructor:

DUM5.PAS	PETE0.PAS	PETE1.PAS
PETE2.PAS	PETE3.PAS	PETE4.PAS

To modify the SPOS program it is only necessary to

Appendix C: Modifying the AFIT Relational Database

The AFIT relational database consists of two separate programs, the DBMGR program and the SPOS program. DBMGR provides the data definition facility and SPOS provides the data manipulation. The following discussion will tell how both programs can be modified.

DBMGR Modification

The DBMGR program consists of five modules that are linked together to form the complete program. Each module is contained in a file and must be compiled before it is linked. The following are the names of the necessary files:

```
DBMGR.PAS                COMON4.PAS               DBDUM1.PAS
DBDUM2.PAS               QUIT2.PAS                COMMON.DEC
COMMON.PRC
```

The files COMMON.DEC and COMMON.PRC are included in the list of necessary files because these files are "include" files for all of the modules. They provide the definition of the necessary data structures used in the code.

To modify the DBMGR program, find the module that needs to be modified and perform the modifications to the file. Then compile the module. The compilation will produce a *.ERL file. This is a relocatable object code file. Before the modules can be linked to provide the DBMGR program a *.ERL file is needed for each *.PAS file. After all of the modules are successfully compiled, the following command is used to link the modules to create the DBMGR program.

```
A:LINKMT DBMGR,COMON4,DBDUM1,DBDUM2,QUIT2,A:FPREALS/S,A:PASLIB/S
```

The combination of the B-tree and leaf page structure means that all the attributes defined to be of a certain domain have their values stored in a common access structure. This also means that each domain defined has its own access structure associated with it except if the domain is defined to be of type, text. Since every attribute value (except those defined to be of domain type text) is stored in an access structure a tuple of relation can be found using any of its attributes not just the key attributes as in some database systems.

tuples that the contain an attribute with the value of the key. The B-tree - leaf page structure is called the access structure in this document. Figure B-2 contains an example of the access structure.

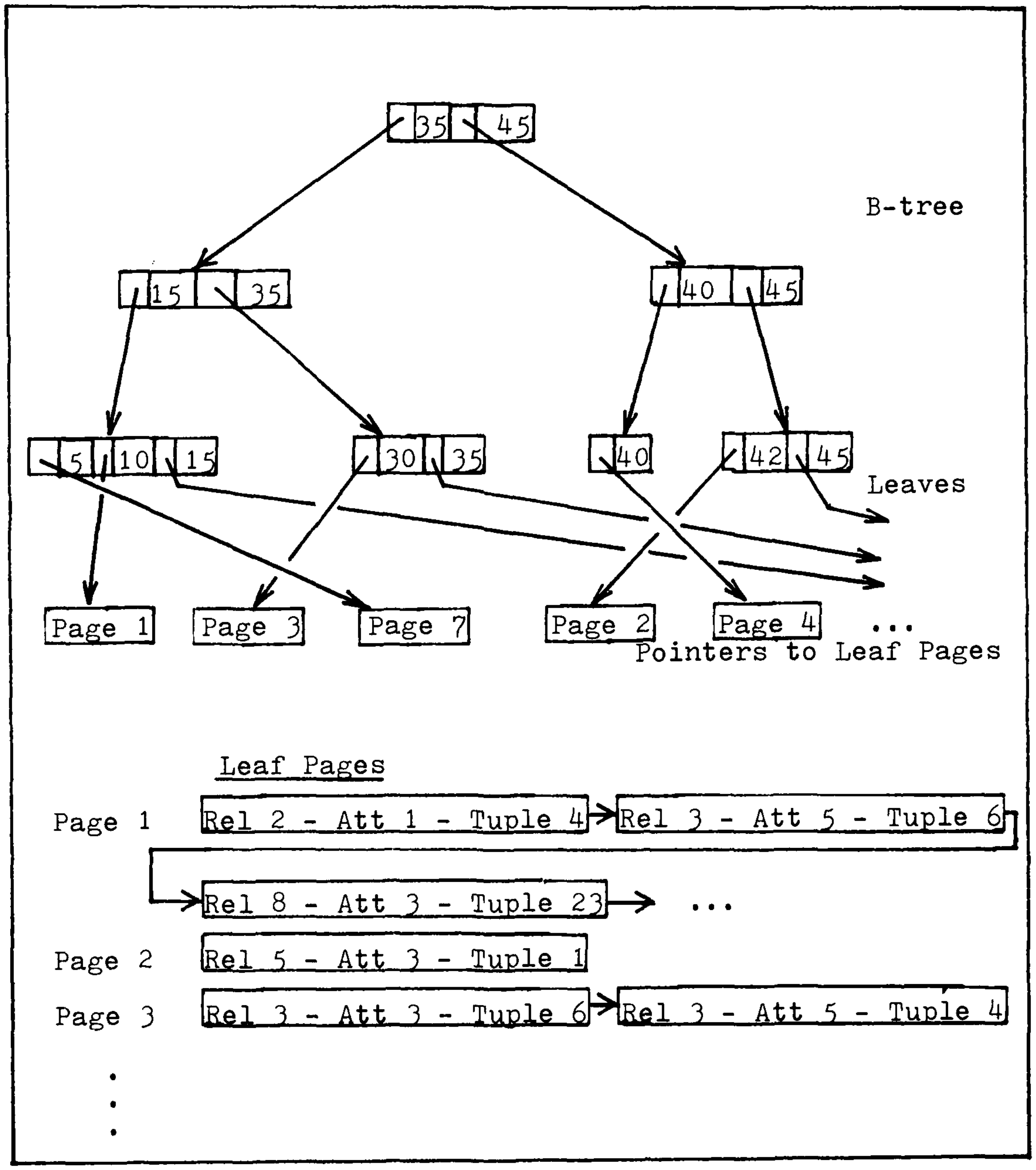


Figure B-2. Example of the Access Structure

Bibliography

1. Haerder, Theo. "Implementing a Generalized Access Path Structure for a Relational Database System," ACM Transaction on Database Systems, 3: 285-298 (1978).

2. Mau, James D. Implementation of a Pedagogical Relational Database System on the LSI-11 Microcomputer, MS Thesis GCS/EE/81D-13. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1981.

3. Kim, Won. "Relational Database Systems," Computing Surveys, 11: 185-211 (September 1979).

4. Raeth, Peter G. An Implementation of a Co-ordinating Operator Constructor. School of Engineering, Air Force Institute of Technology. Wright Patterson AFB, OH, 1979 Unpublished Project Report.

5. Rodgers, 2LT Linda M. The Continued Design and Implementation of a Relational Database System, MS Thesis GCS/EE/82-29. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982 (AD-A124 927).

6. Roth, 2LT Mark A. The Design and Implementation of a Pedagogical Relational Database System, MS Thesis GCS/EE/79-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1979 (AD-A080 240).

7. Smith, John Miles and Philip Yen-Tang Chang, "Optimizing the Performance of a Relational Algebra Database Interface," Communications of the ACM, 18: 568-579 (October 1975).

Vita

Timothy G. Kearns was born on 11 September 1952 in Rushville, Nebraska. He graduated from high school in Rushville, Nebraska in 1970 and attended Chadron State College, Chadron, Nebraska. He received a Bachelor of Science in Math Education in 1974. He completed OTS in September of 1979, receiving a commission in the United States Air Force. From October of 1979 through May of 1984 he worked at HQ AFLC, Wright-Patterson AFB, Ohio, as a computer programmer and software engineer. He entered the School of Engineering, Air Force Institute of Technology, in May 1983.

Permanent address: HC 65, Box 77
Rushville, NE 69360

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE:

3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/84D-12

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: School of Engineering

6b. OFFICE SYMBOL *(If applicable)*: AFIT/ENG

7a. NAME OF MONITORING ORGANIZATION:

6c. ADDRESS *(City, State and ZIP Code)*: Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433

7b. ADDRESS *(City, State and ZIP Code)*:

8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center

8b. OFFICE SYMBOL *(If applicable)*: RADC/CO

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

8c. ADDRESS *(City, State and ZIP Code)*: Griffiss AFB, New York 13441

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

11. TITLE *(Include Security Classification)*: See Box 19

12. PERSONAL AUTHOR(S): Timothy G. Kearns, B.S., Capt, USAF

13a. TYPE OF REPORT: MS Thesis

13b. TIME COVERED: FROM ________ TO ________

14. DATE OF REPORT *(Yr., Mo., Day)*: 84 Dec

15. PAGE COUNT: 125

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES

FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*: Relational Database, Data Bases, Formats, Microcomputers, Access Structure Design, Data Management, Theses, Relational Operators, Management Info Systems.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Title: IMPLEMENTATION AND ANALYSIS OF A MICROCOMPUTER BASED RELATIONAL DATABASE SYSTEM

Approved for public release: IAW AFR [illegible]
LYNN [illegible] [illegible] 21 Feb 85

Thesis Advisor: Dr. Thomas C. Hartrum

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED [✓ XX] SAME AS RPT. ☐ DTIC USERS ☐

21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL: Thomas C. Hartrum, Ph.D.

22b. TELEPHONE NUMBER *(Include Area Code)*: 513-255-3576

22c. OFFICE SYMBOL: AFIT/ENG

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

The single processor optimized relational database system is a database system designed and implemented for teaching and research purposes at the Air Force Institute of Technology. The system was originally designed and partially implemented by Mark A. Roth in 1979. The design and implementation was continued by James Mau in 1981 and Linda M. Rodgers in 1982. To complete the implementation of the relational database system an investigation of the design and implementation of the previous research efforts was done. Additional research was done to explore possible designs and implementations of access structures and possible methods to implement the relational operators.

With this background, a structured design was completed for the access structure and the relational operators. Once this was accomplished, the low level access structure was implemented and tested, providing the capability to insert, delete and modify data in the relational database system. Finally, some of the relational operators were implemented and tested providing an operational relational database system.

Originator-supplied keywords included. [illegible]